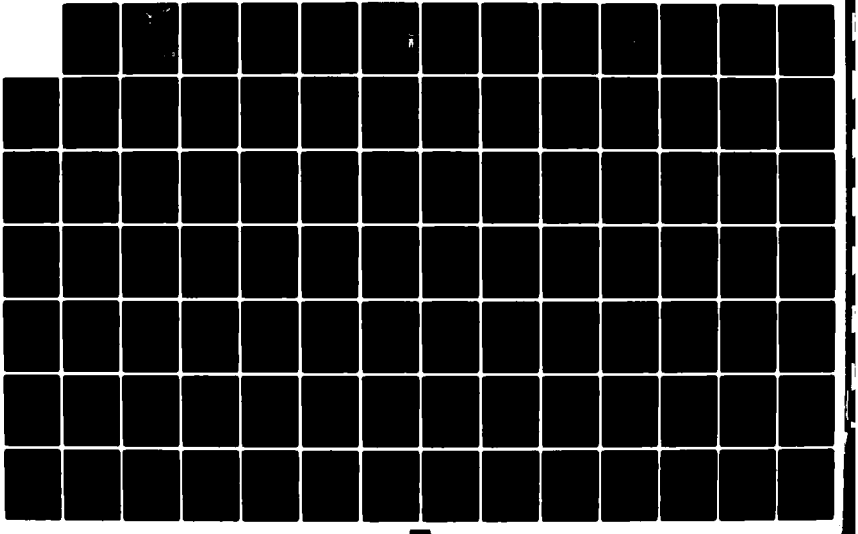
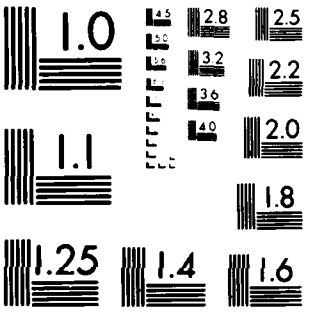
AD-A123 052 A PRELIMINARY EXPLORATION INTO A STRUCTURED APPROACH TO 1/2
SOFTWARE DEVELOPM..(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF SYST.. O H RICHARDS
UNCLASSIFIED DEC 82 AFIT-LSSR-28-82 F/G 9/2 NL





DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

A PRELIMINARY EXPLORATION INTO A STRUCTURED APPROACH TO SOFTWARE DEVELOPMENT COST ESTIMATION

Otis H. Richards, Jr., Captain, USAF

LSSR 28-82

The contents of the document are technically accurate, and no sensitive items, detrimental ideas, or deleterious information are contained therein. Furthermore, the views expressed in the document are those of the author(s) and do not necessarily reflect the views of the School of Systems and Logistics, the Air University, the Air Training Command, the United States Air Force, or the Department of Defense.

AFIT Control Number LSSR 28-82

AFIT RESEARCH ASSESSMENT

The purpose of this questionnaire is to determine the potential for current and future applications of AFIT thesis research. Please return completed questionnaires to: AFIT/LSH, Wright-Patterson AFB, Ohio 45433.

1. Did this research contribute to a current Air Force project?

   a. Yes b. No

2. Do you believe this research topic is significant enough that it would have been researched (or contracted) by your organization or another agency if AFIT had not researched it?

   a. Yes b. No

3. The benefits of AFIT research can often be expressed by the equivalent value that your agency received by virtue of AFIT performing the research. Can you estimate what this research would have cost if it had been accomplished under contract or if it had been done in-house in terms of manpower and/or dollars?

   a. Man-years _____ $ _____ (Contract).

   b. Man-years _____ $ _____ (In-house).

4. Often it is not possible to attach equivalent dollar values to research, although the results of the research may, in fact, be important. Whether or not you were able to establish an equivalent value for this research (3 above), what is your estimate of its significance?

   a. Highly Significant b. Significant c. Slightly Significant d. Of No Significance

5. Comments:

Name and Grade _____ Position _____

Organization _____ Location _____

FOLD DOWN ON OUTSIDE - SEAL WITH TAPE

AFIT/ LSH
WRIGHT-PATTERSON AFB OH 45433

OFFICIAL BUSINESS
PENALTY FOR PRIVATE USE, $300

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 73236 WASHINGTON D.C.

POSTAGE WILL BE PAID BY ADDRESSEE

AFIT/ DAA
Wright-Patterson AFB OH 45433

FOLD IN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: LSSR 28-82	2. GOVT ACCESSION NO.: AD-A123052
	3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): A PRELIMINARY EXPLORATION INTO A STRUCTURED APPROACH TO SOFTWARE DEVELOPMENT COST ESTIMATION	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Otis H. Richards, Jr., Captain, USAF	8. CONTRACT OR GRANT NUMBER(s):
9. PERFORMING ORGANIZATION NAME AND ADDRESS: School of Systems and Logistics, Air Force Institute of Technology, WPAFB OH	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Department of Communication and Humanities, AFIT/LSH, WPAFB OH 45433	12. REPORT DATE: December 1982
	13. NUMBER OF PAGES: 113
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES
Approved for public release: IAW AFR 190-17.
Lynn E. Wolaver
Dean for Research and Professional Development
Air Force Institute of Technology (AFIT)
Wright-Patterson AFB OH 45433
16 DEC 1982

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Software Structure
Software Engineering
Software Costing
Software Development Management
Structured Analysis

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
Thesis Chairman: Harold W. Carter, Lt Col, USAF

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Difficulties in planning and scheduling of software development are inherent in the nature of large scale software design projects, and are recognized to be causes of overruns in developmental dollar and manpower costs. The author focuses on improving this undesirable cost behavior by exploring a prospective basis for software development cost estimation that is compatible to the planning and design functions of a project's lower level managers. The author develops a concept that software controls a set of objects to behave in accordance with a desired behavioral pattern defined in a set of design specifications. If the structure of the desirable behavior is a good predictor of the pending software's structure, the author reasons that the structure of the desirable behavior might also serve as a basis of cost estimations techniques that are more consistent with the planning and scheduling needs of intermediate project managers. The author develops a model for establishing a functional structure to the object's desirable behavior, develops software using his model, and measures the correlation between the software's structure and the behavioral structure projected by his model. He concludes that the structure of the desired behavior is insufficient as a single-factor predictor of software structure. There are factors which are unique to software, such as optimization of software code, which cause software structure to differ from the structure predicted by that of the desired behavior.

LSSR 28-82

A PRELIMINARY EXPLORATION INTO A STRUCTURED APPROACH TO SOFTWARE COST ESTIMATION

A Thesis

Presented to the Faculty of the School of Systems and Logistics

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Systems Management

By

Otis H. Richards, Jr., BS
Captain, USAF

December 1982

This thesis, written by

Captain Otis H. Richards. Jr.,

has been accepted by the undersigned on behalf of the faculty of the School of Systems and Logistics in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS MANAGEMENT

DATE: 17 December 1982.

COMMITTEE CHAIRMAN

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION AND PROBLEM STATEMENT

THE NATURE OF THE PROBLEM

Since the advent of computers, the escalating volume of prospective applications has evolved a corresponding expansion in the size and scope of computer software. The size and scope of modern "heavy industry" programming overwhelmingly surpasses any individual's level of detail comprehension (1). The requisite shift in software development procedures from an individual focus to a group focus has motivated a revolution in developmental techniques.

The prominent impetus for the revolution has been the disproportionately escalating monetary and productivity losses encountered when state-of-the-art small scale techniques have been applied to large scale projects. A 1979 Government Accounting Office (GAO) study of nine software acquisitions revealed that $1.95 million out of the total $6.3 million purchases (31%) was paid to contractors who failed to deliver contracted software (2). The costs of poor performance were attributed significantly more to deficiencies in management planning and control of development activities than to technology factors (3:727). Experts attribute many of the difficulties in managing software development to complications in both establishing

intermediate progress targets and evaluating progress status during the development process (2). The potential for losses attributable to deficient management practices and techniques becomes highlighted when one considers the size and growth implications of the programming industry. Software costs in 1976 were estimated at around $20 billion (4, 2). In 1980, software costs totaled $40 billion, or about two percent of America's Gross National Product. The rise in software activity reflected by these cost figures is expected to continue (3, 4, 2).

Labor considerations dominate other software development cost drivers. The programming task entails decomposing highly complex processes into elemental transformations and data entities which are to undergo transformation, then resynthesizing them into some rigorous artificial format. Although this tedious mental labor founds software cost and quality phenomena, a systematic understanding of its rudimentary constructs is conspicuously lacking (2).

BACKGROUND

Correlational vs Theoretical Discovery.

Recent undertakings to project software development costs follow one of two different avenues for advancement. Empirically derived models dominate existing cost forecasting procedures. Based on their accumulated knowledge, experts identify and define cost drivers and then apply rigorous statistical and correlational analyses of

completed software development projects to refine, support, and expand their perceptions. Boehm, McCall, Brown and Gilb have all substantially contributed to this avenue of progress. The theoretical avenue, and the one we will pursue, engages rigorous logic to explore levels of abstraction frequently too subtle or conceptually interdependent to gain discovery through correlational techniques. Parties interested in advancement of software engineering technology recognize this avenue to be in need of "travelers". Of late, Halstead, McCabe, Basili and Reiter, Myers, and Belady have all evolved notoriety by expanding the theoretical foundation of software engineering (2).

Lines of Coding Estimates.

Current techniques for projecting software manpower rely largely upon the accuracy of an estimated number of completed Lines of Code (LOC) projected to comprise the finished software. While the accuracy of these LOC-based estimates may be adequate for contractual purposes, they exhibit drawbacks for use in providing managers with intermediate progress goals. In the first place, LOC estimates are estimates of a lump-sum quantity of code. The estimates provide no mechanism to assist in effective partitioning of the project. Without the milestones provided by partitions against which to measure accomplishment, managers lack critical information with which to assess a rate of progress. Basing an estimate for manpower require-

ments remaining in a development project upon aggregate LOC could conceivably frustrate the accuracy of any intermediate progress goals, since such an extrapolation to small levels of aggregation exceeds the intended scope and purpose of LOC estimates. Limitations in the scope of LOC estimates preclude their use by managers as tools with which to devise and enforce schedules at their scope of involvement. In the absence of an estimator more congruent with scheduling purposes, economies attributable to the scheduling of subordinated tasks will remain unachievable; and, the status-quo of escalating software development costs can be expected to continue.

Secondly, the actual volume of LOC comprising a program is heavily dependent upon the skills and abilities of its authors. For example, one software developer may be able to complete a task in four hours, producing 100 LOC. Another might require six hours and produce 200 LOC to accomplish the same task, while another may require five hours to accomplish the same task in 80 LOC. Indeed, even the same developer who achieved the task in 80 LOC on Friday may have labored four and a half hours and produced 90 LOC were he to have performed the task instead on Monday. There is evidence that the volume of LOC actually comprising a piece of software is based on multiple independent factors, and is extensively dependent on programmer aptitude (6).

Thirdly, for the scope and purpose of managing software

development, the LOC concept embodies no indication of work efficiency. Any LOC estimate inherently incorporates the classical difference between productive achievement and the discrete but unrelated quantity of work effort leading to that achievement. LOC is an achievement; a product. Yet an estimate for LOC is used to project the manpower effort needed to achieve that product. In the previous example, as in reality (6), the LOC finally produced by each programmer did not accurately reflect the time they consumed. While a discrete relationship between the volume of LOC and the volume of work effort pursuant to their realization appears impracticable, a probabilistic relationship may exist. However, the state of development of any rigorous relationship between LOC and effort hardly appears adequate to provide middle managers with useful targets for either intermediate progress goals or performance measurement of programmers.

It is evident that these innate properties of the LOC basis at least exacerbate, if not cause, fundamental and costly difficulties experienced by middle managers in establishing intermediate project progress. To enhance the usefulness of manpower estimates towards intermediate progress management, a basis which better overcomes these drawbacks must be derived. A basis which better accommodates partitioning may be readily available. This thesis will explore one alternative vehicle by which to accommodate partitioning.

Functional Modularity (FM).

Upon preliminary exploration, a prominent issue which shows promise as a basis for manpower estimations is Functional Modularity (FM). Functional modularity refers to the concept that any function which is to be performed by software can be decomposed into aggregates of simpler functions (figure 1). It is founded in an analytical process whereby a software project is iteratively dissected into subordinated functions, exposing progressively deeper layers of subordination and dependency. Careful analysis of prospective modular configurations seeks to expose that set of subordinated functions which exhibit the highest degree of independence from each other. Once an acceptably independent set of subordinated "simpler" functions has been isolated, each will be treated in one of two ways. Each will be either iteratively analyzed into further subordinated functions, or programmed into a functional software module if it lacks the complexity to warrant further analysis. When the analysis concludes, a structure of elemental functions and their interrelationships lay exposed to guide ensuing software construction activity. The software engineering discipline encompasses several techniques which promote FM concept. Among these techniques are Data Flow Diagrams, Data Structure Diagrams, and Flow Charts (5, 7).

Prospective Managerial Applications of FM.

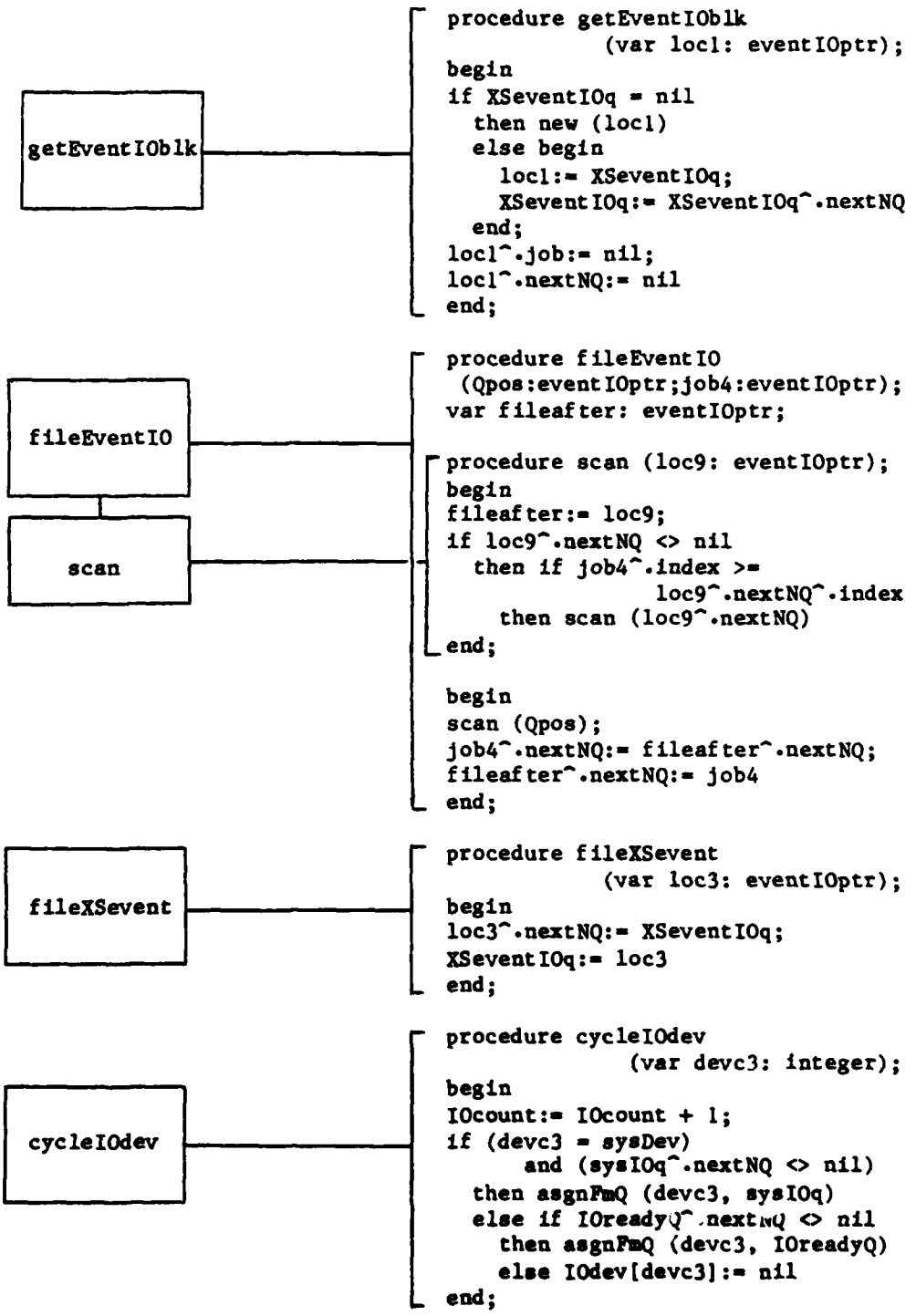


Figure 1 - Software Modularity

Advantages of functional modularity over LOC may provide a superior basis for cost estimation for the developmental middle manager's use. Let us define a modular projection to be a count of those software modules derived from a modular structural analysis of some type. A difference between a LOC projection of a software development project and a modular projection of the same project is that a modular estimation is based on a smaller number of larger units (a module is "larger than" a line of code). This may yield some refinement in the accuracy of statistical methods by reducing the range of deviations. FM should not, however, exhibit any improvement in sensitivity to prediction inaccuracies induced by personal skill differences.

There is a more important enhancement in modular projections over LOC projections. Though both projections are based on a concrete referent (a module and a line of code), the LOC referent displays no functional, or logical, value. A module does. The functional information contained in modular referents could enhance both the manager's and the designer's technical comprehension of the design project. Increased comprehension could in turn improve both the accuracy of estimates and the quality of managerial decision. Also, because of the additional information provided by the functionality of modules, the uncertainty prevalent in software construction projects should be reduced below that of a comparable project without the information.

Partitioning a software project into modules also shows intuitive prospects the scheduling and performance measurement aspects of middle management. In scheduling, the additional information provided by functionality can highlight obscure details of specific interrelationships which might otherwise remain undiscovered. This might provide greater scheduling accuracy. FM may also prove useful in performance measurement of designers and programmers. If the structure of the system into which the software will be placed can be deduced, then a comparison between the system structure and the software modular structure might serve as a basis for a performance measurement against the software design. This prospective application of functional structure to performance measurement highlights an additional design concern. With FM, there exists a set of design alternatives addressing optimization alternatives for the modularization of software. Modularization is the activity of partitioning functions into software modules.

The total developmental workload in a software design project should not be substantially extended by management use of FM analysis. In development projects employing state-of-the-art software engineering doctrine, FM is already incorporated into the initial phases of project design (3, 7). The only additional effort should be that of adapting design FM for management use.

THESIS DIRECTION

Of concern to this thesis is the basis upon which software modularization design decisions are made. Existing software engineering literature focuses heavily upon data structure as the foundation for software modularization. The acceptance by structured software design enthusiasts of data structure as a modularization foundation seems widespread (5, 7), but without other than intuitive justification. The link between data structure and software modularity may well have come as a natural consequence of an analytical focus of these enthusiasts. Software examples referenced by authors of structured design literature tended heavily towards data-processing software (5, 7).

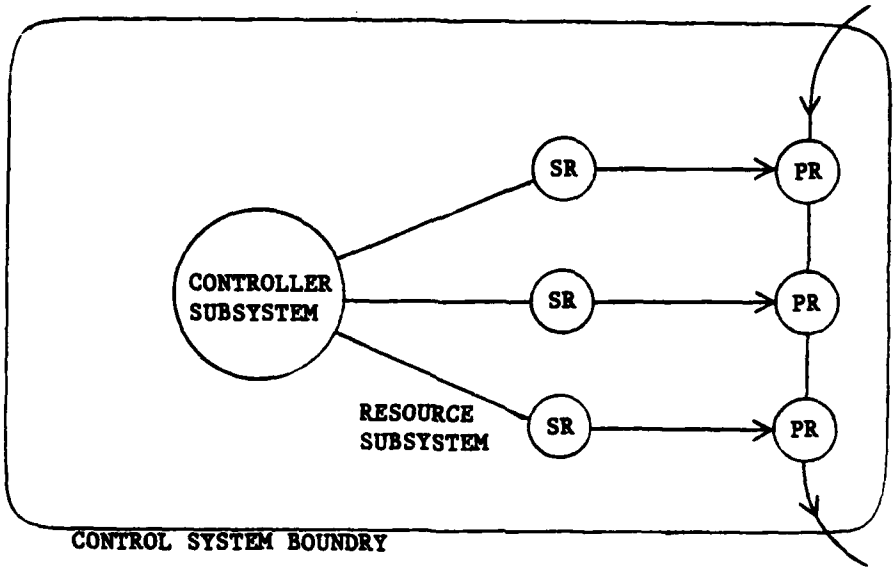
Perhaps there are foundations other than data structure upon which to base modularization decisions. The data-processing-type software prevalent in literature processes an abstract entity called data. However, other categories of software deal with concrete entities. Examples of such software manipulate weapon subsystems within modern military machinery, and control complex industrial automation equipment. Observing that the structure of data-processing software is founded in the structure of the "data" entity it manipulates, one might suspect the existence of another modularization foundation more closely aligned with the structures of concrete entities.

Control Systems.

To begin exploration, consider the function of software. Software controls something. Whether software's application is to industrial robots, video displays, pure data or something even more abstract - software is involved as that something's controller. Control is the fundamental function, and common bond, of software.

A general controlled system is comprised of a controller and resources, which are controlled (figure 2). The resources a controlled system is designed to control are its Primary Resources (PRs). In general, though not always, PRs are not directly controlled directly by the controller. Other resources called Secondary Resources (SRs) intervene in the control chain. SRs, in general, are controlled directly by the controller; and directly control the PRs according to the directions of the controller. A Software Controlled Control System (SCCS) differs from a general controlled system in that its controller contains software.

The resource subsystem of a controlled system can be considered independently of the controller. For example, consider a control system to adjust room temperature. A coal furnace might be a resource subsystem which accomplishes the heating objective. Coal and air are burned to produce heat. Coal would constitute one PR and air another. Interrelationships between the resources in the resource subsystem would cause each to exhibit a



NOTE: The controlled system is a SCCS if its controller subsystem embodies software.

Figure 2 - Control System

deterministic behavioral response to the activity of the others. However, only a subset of possible responses may be desirable. In the example, the coal and air would just sit there. This is an undesirable behavioral response in terms of the stated objective. The function of control is to alter the behavioral response of the resource subsystem to assure that each individual resource's behavioral response is desirablely consistent with an objective. Hence, the controller's influence upon the resource subsystem must both ensure occurrences of desirable behavior and suppress occurrences of undesirable behavior.

PROBLEM STATEMENT AND SOLUTION APPROACH

Let us define the structure of a resource subsystem to be a mapping of the network of interrelationships existing among its various resource elements. It is evident that, in a controlled system, the structure of a resource subsystem influences the design of its controller subsystem. Of concern to this thesis is the influence that a SCCS's resource subsystem structure exhibits over the modular structure of software in its controller. This thesis will investigate the strength of this influence. Of immediate concern is the influence of the resource subsystem's structure upon the selection of those design alternatives pertinent to modularization during software's construction phases. Of underlying concern is the prospect of using the resource subsystem's structure of the resource subsystem to

acquire a projection of the modular structure of software, and to exploit this structure as a foundation for an array of tools and techniques useful to the intermediate managers of software development projects.

In support of an exploration of the stated immediate concern, the following hypotheses are proposed. Let us preface the hypotheses with the following definitions.

A software-unit is the physical coding which accomplishes a single - i. e., a "unit" of - logical function (figure 3). Generally, several line of coding make up a software-unit. A software-unit is the most fundamental "physical" entity defined by logical contents. Functionally equivalent software-units accomplish identical logical functions, though their physical compositions are not necessarily identical. In software development, design alternatives manifest themselves as functionally equivalent software-units.

A software module is that collection of software coding which is bonded by the software's designer into a single unit, and can be referred to from elsewhere in the software code as a unit. Generally, a module is a collection of software-units focused about some central function. It is itself a software-unit. It is generally assigned a unique name, or identifier; and can be entered into execution by a reference to that name. "Module" is a referent for the "physical" container of functional, or logical, contents.

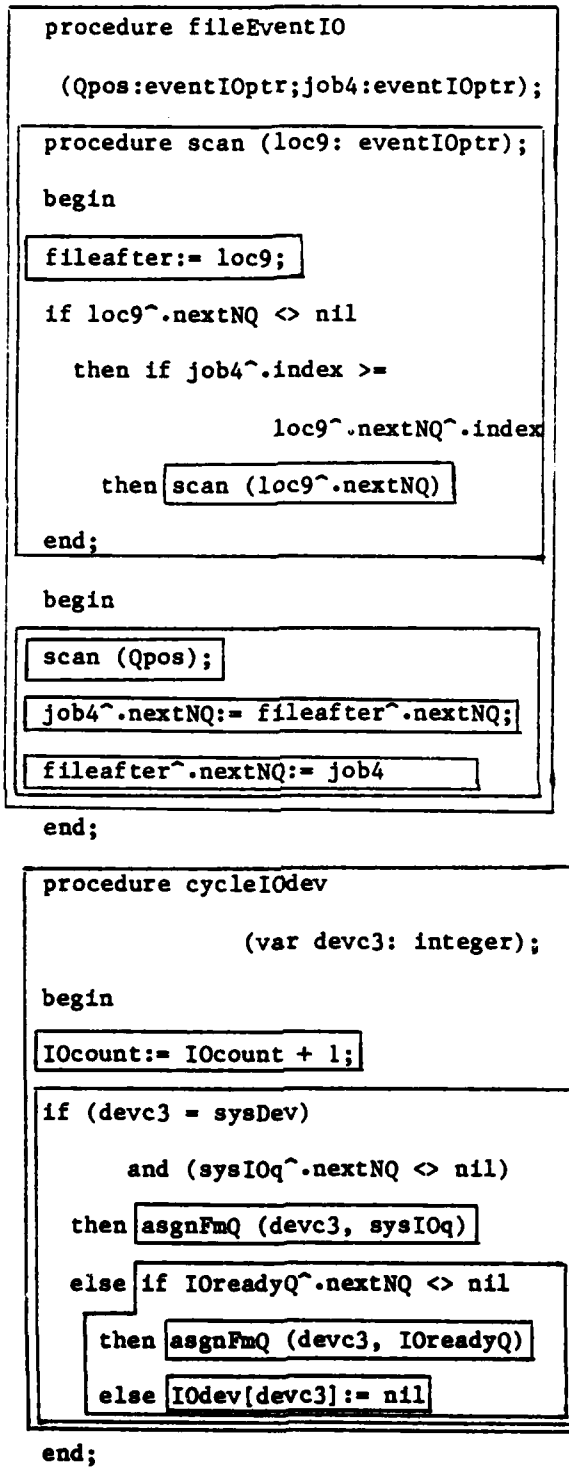


Figure 3 - Software-Units. Software units are boxed.

The modular structure of a software program refers to the skeletal-type function performed upon the software program by its constituent modules. It is the collection of functional interrelationships that bind the modules of a program together into the program unit.

HYPOTHESIS I: In a SCCS, the structure of the resource subsystem is sufficiently predictive of the controlling software's modular structure to be used as a foundation for an as yet undeveloped body of software costing and design management tools and techniques.

For the purposes of this thesis, we shall define "sufficiently predictive" to be a deterministic relationship.

HYPOTHESIS II: In a SCCS, there exists a deterministic relationship between the structural elements of the resource subsystem and the software modules in the controlling software.

As a vehicle with which to initiate the exploration, we shall examine the prospects that the simplest possible deterministic relationship exists:

Hypothesis II is not worded in its final form. A major faction of this thesis is the development of a particular model which reifies the concept of structure in the resource subsystem. Hypothesis II will be refined after the model is proposed to reflect appropriate terminology

from the model.

Overview of Solution Approach.

The remainder of this section delineates fundamental constraints upon the investigation. The next section, Concept Definition, begins by defining concepts fundamental to the SCCS. The initial discussion of the resource subsystem in the SCCS is posed from a general systems perspective, independent of software considerations. Though this approach may appear tedious, the perspectives so introduced constitute a foundation of system elements which can be organized into a general resource subsystem structure. Continued discussion then exposes from among those system elements an indigenous structure of general functional interrelationships. As the focus then shifts to SCCSs, a model called a transformation state diagram is developed which projects the indigenous structure into a visible form.

Section III, Concept Demonstration, is somewhat of an extension to the theory advance in section II, in that the theory posed in section II is summarized into a three-step process for analyzing the structure of a resource subsystem and generating its transformation state diagram. In conjunction with the summary, each step of the analytical process is demonstrated as the step is presented using a sample software development project. The analysis of the sample project produces a transformation state diagram of the

project's resource subsystem structure which was employed by the author in developing the software of Appendix A.

In section IV, Concept Validation, the software of Appendix A is compared to the transformation state diagram to determine the degree of correlation between the software's modular structure and the resource subsystem's structure as given in the transformation state diagram. Section IV also embodies an analysis of the results. Conclusions and recommendations for further research follow in section V.

Limitations of Solution Approach.

Further discussion will adhere to the following constraints.

(1) The resource subsystem for which control software is under development has been extensively, but not necessarily completely, developed before software development activities begin. The contents, interrelationships, and desired behavior of the resource subsystem are substantially known.

(2) The discussion proceeds from the assumption that the software will be developed in a Higher Order programming Language (HOL). This constraint does not imply that the discussion is not pertinent to other languages. It merely emphasizes that many of the correlations drawn between a HOL construct and the resource subsystem will be less correlated to specific language constructs in the other language.

(3) As a caution to readers possessing a strong background in the conventionally accepted lexicon of digital control or related formal disciplines, be advised that the phraseology used herein is unique to this author. This thesis is focused on software, but relies heavily on the author's perceptions of control systems - a discipline in which the author lacks formal training. Connotations of key words used herein may conflict with those of the formal discipline. Additionally, such a systems-oriented approach to software design lacks a substantial body of knowledge upon which to draw.

(4) The emphasis herein lies on synchronization aspects of software, as compared to the adjustment aspects. Synchronization refers to the act of sequencing multiple individual resource elements into a functional order of operation. Synchronization principally involves determining the appropriate instant at which to instigate or terminate interrelationships between resources. It deals with interrelationships in a binary fashion. Synchronization occurs, for example, when the shutter release of a camera is depressed. Many other activities within the camera are synchronized to that act of depressing the shutter release. Adjustment refers to the refinements in the nature of these interrelationships during the period between their instigation and termination. An adjustment on a shutter release, for example, might be adjusting how much force is needed to depress it. Adjustment deals with interrelationships as a

continuum, as opposed to the on-or-off binary nature of synchronization. Adjustment is a conceptual extension of synchronization; but, time constraints upon the author necessitate an emphasis upon the more fundamental synchronization aspect. The author speculates that further exploration of ramifications of the distinction between synchronization and adjustment, as it applies to software design, will associate synchronization with control algorithms and adjustment with response algorithms.

(5) For this thesis, feedback is considered to be a phenomena implicit in a system's sensing subsystem (discussed later) and software. Other than the following explanation of the author's intent behind the proceeding sentence, it is not explicitly addressed.

Feedback occurs when the controller senses what changes have occurred in the resource subsystem as a result of previous controller manipulations of the resources therein. For closed-loop controlled systems, feedback behavior is almost always a critical determiner of overall system performance. In these systems, the time period required for the controller to act (i.e., sense the resource subsystem, formulate a controller reaction, and impose its reaction upon the resources) approaches the time period required for the resource subsystem to attain a permanently undesirable configuration (i.e., a system configuration from which return to a desirable configuration is impossible). In such controlled systems, especially precise and detailed

consideration must be paid to feedback phenomena to preclude the occurrence of an irrecoverable resource subsystem configuration, such as destruction of parts of the resource subsystem.

However, explicit attention to feedback behavior is not always required. This thesis limits its consideration to controlled systems in which the controller's response time period is comfortablely smaller than the time period required for the resource subsystem to achieve an irrecoverable configuration. By eliminating from consideration controlled systems which are sensitive to the details of feedback phenomena, feedback phenomena can be disregarded as a software design consideration. This feedback limitation seems appropriate in light of the preliminary nature of this thesis.

CHAPTER II

CONCEPT DEFINITION

FUNDAMENTAL CONCEPTS OF RESOURCE BEHAVIOR

The concept of control carries with it several implications. First is the existence of "things" to be controlled. Second, there exists some underlying objective to be achieved as the result of control activity. Of course, the presence of an objective implies that something is sought which is not currently available; and thus foretells of an impending transformation, or change, of what is available *into what is desired*. The occurrence of a transformation reveals the presence of some type of force or energy, since nothing can be altered without an energy source. Third, the "things" and the transformations they must undergo must be sufficiently structured to permit the introduction of a control mechanism. And finally, control implies the existence of a controller.

BASIC COMPONENTS OF THE TRANSFORMATION PROCESS

Resources.

The transformation process is a conceptualization of the changing of "what is available into what is desired" (figure 4). Each "thing" which changes during the course of a transformation is a resource. Basically, anything

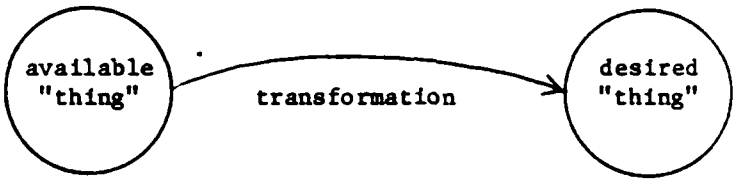


Figure 4 - The Transformation Process

which can be conceptualized as an entity would be a resource. This concept of resources accommodates a large range of abstraction; ranging from concrete resources, such as a baseball, to abstract resources, such as utility. The English Language referent for a resource would be a noun. An example of a resource would be coal.

Resources fit into categories. Categorized according to the nature of the transformation a resource undergoes, a resource is either consumable or nonconsumable. Consumable resources change their form irrecoverably during the course of transformation. Consider for example a coal furnace. Coal and oxygen from the air are consumable resources. Consumable resources are characteristically transitory to the system. They transit across the system boundary from without to enter the control system, and transit across the

boundary (in some other form - frequently aggregated with other consumables) to exit. Nonconsumable resources recover their pre-transformation form during the course of objective achievement. For example, the vessel containing the burning coal is nonconsumable. These resources are consistent system member: they do not transit the system boundary.

Categorized according to how a resource participates in objective(s) achievement, a resource is either a Primary Resource (PR) or a Secondary Resource (SR) [reference discussion, page 11]. Transformation of the PRs is the objective to which a controlled system is designed. PRs are generally consumable. PRs in the burning coal example would be coal and air. SRs contribute to the transformation objective by performing some type of function upon the PRs. In the coal furnace example, the vessel containing the burning coal is an SR. Among other contributory functions, the vessel provides an influence to retain the PRs in relative positions so that the objective transformation will persist. SRs often embody the energy sources which affect the desired transformation of the PRs. For example, a lathe, which transforms a rectilinear column of metal into a cylindrical column of metal, constitutes a SR. SRs are usually nonconsumables, though not always. When the transformation of a PR cannot be directly controlled by the system's controlling agent, SR(s) are frequently imposed to indirectly achieve control over the transforming PR.

Discussion of this application of SRs is deferred pending introduction of other instrumental concepts.

The fundamental units within a controlled system are its atomic resources. Subsystems of the resource subsystem are groups of one or more resources which share some functional relationship. Resources or resource subsystems which are atomic to the control system are those single resource elements or subsets thereof which are themselves controlled directly or indirectly by the system controller, but which do not embody any component resources which are controlled via a separate interrelationship with the controller (figure 5). For example, consider a card reader. The part of the reader which feeds the cards is activated directly by the controller. The feeder is an atomic resource. The cards themselves are controlled by the feeder mechanism, and are therefore indirectly controlled by the controller. The cards have no more direct interface with the controller. Their only interrelationship to the controller is the indirect relationship just described. The cards also are atomic resources. Information acquired from reading activity is sent to the controller via a separate link, or interrelationship, than the link which activates the feeder. The component which reads the cards and sends information to the controller is an atomic resource. Considered as an assembly, however, the card reader unit is not atomic since it embodies other resources which are linked to the controller via separate

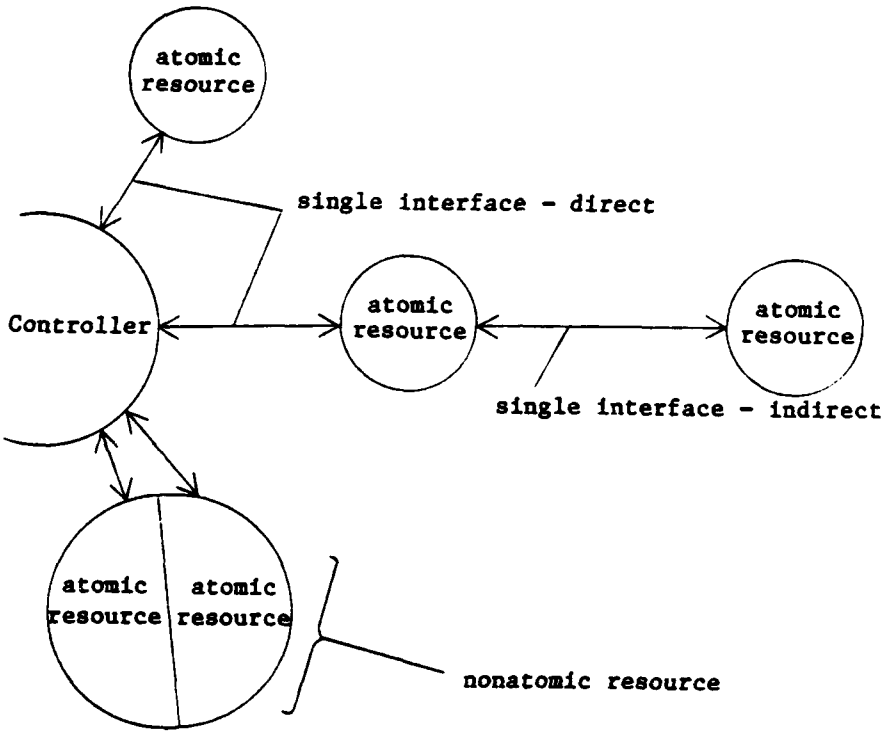


Figure 5 - Atomic Resources

interrelationships.

Resources have attributes. A unit of coal, for example, has a certain hardness, a certain mass, and a delivers a certain quantity of heat energy when burned. Attributes are also nouns in the English language. Some attributes are characteristic of the resource regardless of the resource's membership in a particular system. Hardness and mass are examples. Others, called relative attributes, pertain to a resource, but are precipitated by the presence of the resource in a particular system. Examples of relative attributes would be proximity in time or distance to other resources.

Attributes have values. The unit of coal might have a weight of two ounces, and produce 50 calories of heat per ounce when burned. The values of these attributes describe the resource as adjectives describe the nouns they modify. Values of many attributes of a resource will change over the course of a transformation.

Influences.

Transformation forces are as instrumental to a transformation as are resources. Resources can not transform without a source of energy to drive their transformations. These energies may be naturally embodied within the resources, such as the energy that drives the coal-oxygen oxidation; or they may themselves transit into the system, such electricity through a wall outlet. These

energies manifest themselves as forces which, according to Newton, act equally and oppositely on more than one subject mass. Accepting that resources are masses, it is apparent that any transformational force must act equally and oppositely on at least two resources. Hence, transformational forces become indicators of the interrelationships that exist among the various system resources. This is the foundation for the concept of using resources and transformational forces to construct a structure to a resource subsystem of a controlled system. In this thesis, the term influence is used in lieu of the term force. Two categories of influences are important (figure 6).

The transformational influence refers to that underly-

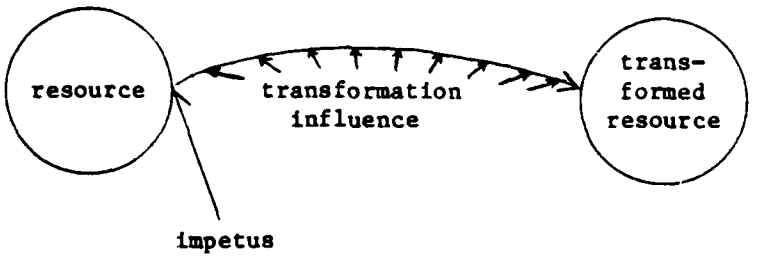


Figure 6 - Transformational Influences

ing force which perpetuates a transformation. More specifically, the presence or existence of the transformational influence perpetuates a dynamic progression of value changes in some subset of a resource's attributes. Transformational influences can be mechanical, electrical, chemical , etc. in nature. In the burning coal example, the transformation which causes oxidation is chemical in nature. Another transformational influence encompassed in the burning coal system is the mechanical influence which causes the air in proximity to the burning coal to be replaced by more air as the chemical activity proceeds.

The second category of influence is called the impetus for a transformation. The impetus is the influence which initiates, or excites, the transformational influence. From the perspective of the atomic resource, its immediate environment within the system must be prepared for transformation as a prelude to its participation in transformation. The impetus refers to the energy which caused the final act of preparation: the act which unleashes the transformational influence. In a coal-air system, the final act of preparation would be the ignition of the coal; but if some more powerful oxidant were used, the final act might be merely introducing the coal and oxidant into proximity of each other. An impetus does not permeate the transformation step as does the transformation influence. It is non-persistent in that its duration is sufficient only to initiate the transformational influence. In

controlled systems, the controller frequently oversees or accomplishes the preparation of resources prerequisite to transformation, including virtually all impetuses. In summary, an impetus can be perceived as the first influence of the transformation process, and the transformational influence as the driving influence of the transformational process.

STRUCTURE OF THE TRANSFORMATION PROCESS

A single occurrence of a transformational influence provides the fundamental building block for resource subsystems. A single occurrence of a transformational influence endures from its initialization by impetus until termination. At the point of initialization, each resource subject to the transformational influence is stable - i. e., their attribute values are unchanging. The configuration of a resource refers to the values exhibited by all its relevant attributes at a single point in the transformation process. The transformational influence is then initiated by impetus, causing the transformational influence to act. During the course of the transformation, the affected resources' attribute values continue to change until the transformational influence terminates. The transformational influence terminates as a result of either exhaustion, such as when all the coal in the burning coal system has been oxidized; or by controlled interference, such as when a controlling agent sprays water on the coal

to starve the burning influence by depriving it of the air resource. Following termination, the configurations of affected resources again stabilize - their attribute values stagnant in the absence of a driving transformational influence. Affected resources will remain in stable configuration until preparation of immediate environment, to include at least another impetus, reoccurs to activate another occurrence of the transformational influence. These stable configurations are the states of a resource. The configuration with which a resource begins transformation, the transformation, and the configuration with which the resource concludes transformation collectively form the fundamental structural element of a resource subsystem. This fundamental structural element is a transformation step.

Thus, a simple transformation step can be envisioned as continuous from one state to another, and discontinuous at its end points (figure 7a). The stable configuration from whence a resource begins its transformation is its initial state. The stable configuration which the resource attains as a result of having undergone transformation is its terminal state.

The concept of a transformation step provides the foundation for a conceptual structure within complex transformations. Complex transformations embody more than one transformation steps chained together to accomplish a single transformation objective (figure 7c). To avoid

confusion among the various transformation objective concepts, we shall define the transformation objective of a single transformation step to be the step objective; while the overall objective of a transformation, simple or complex, to be the final transformation objective. The first initial state of a resource is followed by transformation to a temporary terminal state. The temporary terminal state constitutes the initial state for the next transformation, and so on. These temporary terminal states which are the terminal state for one transformation and the initial state of another transformation are called interim states. A resource's first initial state - i. e., the state from which transformation activity begins progressing towards the final transformation objective - is also called that resource's start state. The resource's last terminal state - i. e., the resource's state upon reaching it final transformation objective - is also called the stop state (figure 7b). During the course of complex transformations, each of the system resources will undergo some sequence of transformations steps. Since a resource subsystem's PR(s) form its central focus, it appears rational to define the structure of a complex transformation to be the sequence of transformation steps through which the PR(s) pass.

TRANSFORMATION ANALYSIS

Analysis seeking to expose the characteristic transformation steps which comprise a more complex transformation

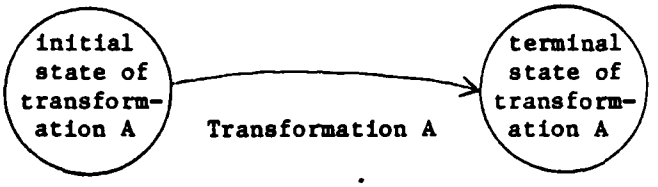


Figure 7a - Simple Transformation

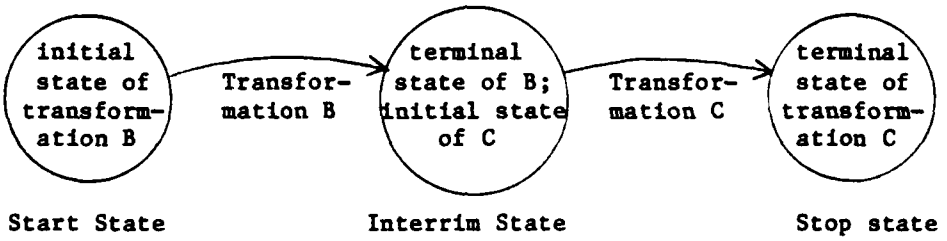
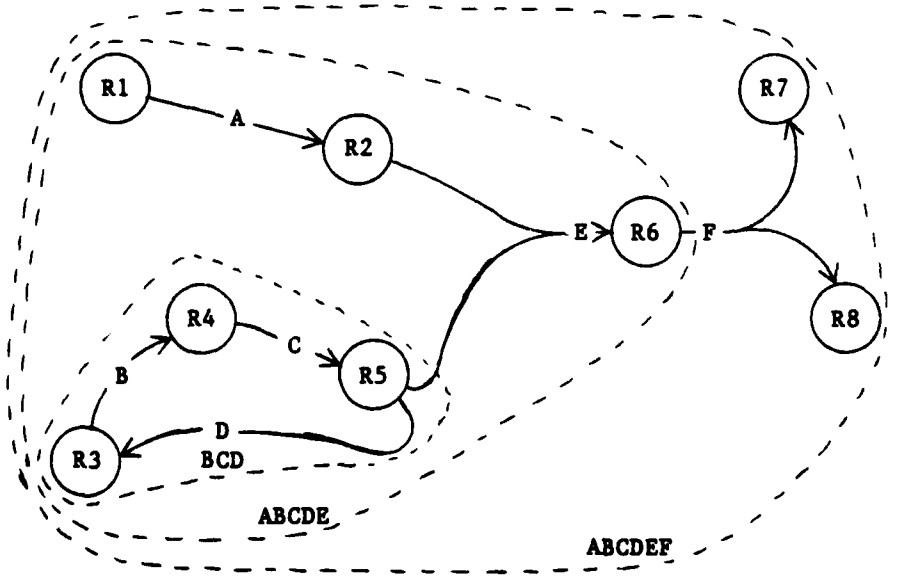


Figure 7b - Complex Transformation



NOTE: A, B, C, D, E, and F are Simple Transformations. BCD, A(BCD)E, and ABCDEF (which is whole system) are Complex Transformations. Resources are given above by R#.

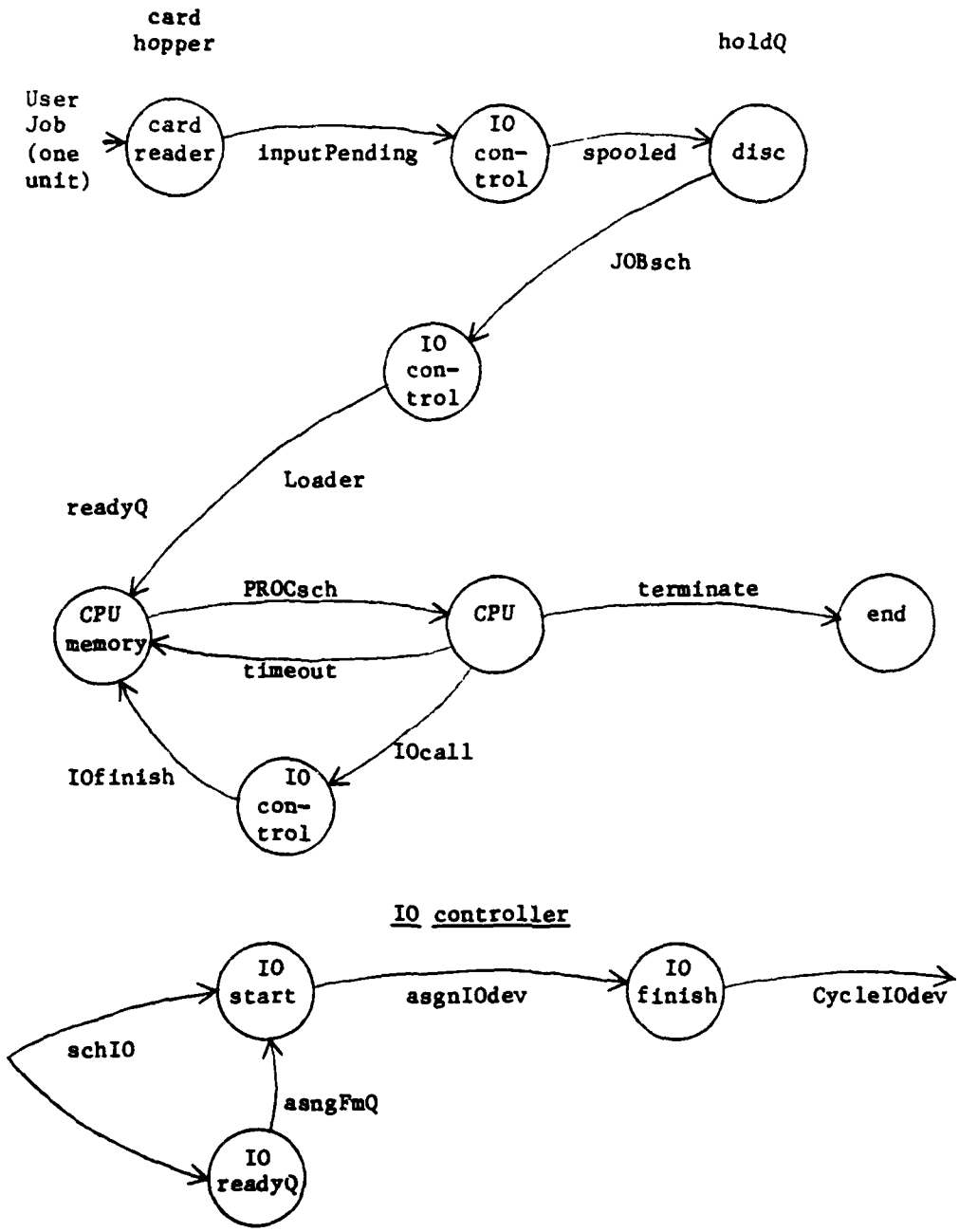
Figure 7c - Aggregation of Simple Transformations into Complex transformations

is called transformation analysis. All the resources of a resource subsystem - each of which might well undergo a complex transformation - are involved in transformation analysis. Transformation analysis seeks to dispurse the common objective about which the resource system is designed into a transformation structure for each resource. Transformation analysis produces a mapping of a resource subsystem's elements and interrelationships suitable to form a transformation state diagram. Figure 8 illustrates the transformation state diagram which we will subsequently develop and apply.

The common objective upon which transformation analysis is based is the design objective. It is the objective upon which the entire control system - not just the resource subsystem - is founded. The design objective performs two prominent functions for transformation analysis.

(1) The designation of PR or SR is assigned to each resource based upon its participation in achieving the design objective. For example, a lump of coal would be a PR were it to be burned to provide heat under a design objective of warming a house. It would be a SR were it to be used as a door stop under the design objective of retaining a door open after the door had been transformed to an open state.

(2) The design objective tends to implicate start states and stop states for the resource subsystem's PRs. For example, if a house is to be heated, the whatever is



NOTE: there are thirteen transformation steps. The arbitrarily assigned names of the transformation steps (not modules) are shown.

Figure 8 - Transformation State Diagram

selected as a PR will have completed its transformation (stop state) when it has transformed into a vehicle by which the house can be heated. If coal is selected in conjunction with this design objective, then coal will be the PR's start state. By implicating the start and stop states of the PRs, start and stop states of the SRs Start and stop states of the SRs are implicated in their specific contributions to the transformation of the PRs. For example, a damper of a coal furnace must be sufficiently open to permit ignition of the PRs.

Impact of Control on Resource Subsystem Structure.

We have developed a tool with which to structure a set of interacting resources. However, our interests lie in that specific set of resources that comprise a resource subsystem of a controlled system. This set of resources bears the distinction that its member resources interrelate with each other, but also interrelate with a controller. Although the specific nature of these interfaces exceeds the scope of this thesis, it is appropriate to examine the impact the introduction of control has upon a resource subsystem's structure.

Control of a transformation is achieved by moderating a transformational influence. For example, to control the speed of a ball rolling down an incline plane, the slope of the incline can be adjusted (figure 9). This moderates the strength of the influence that causes the ball to transform

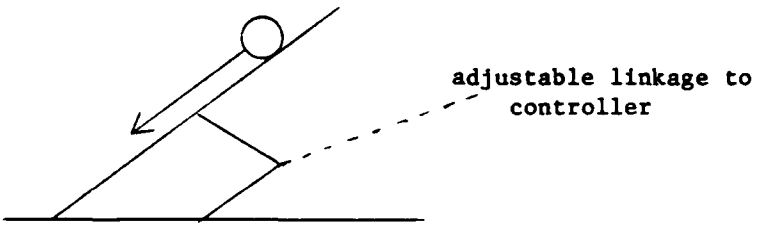


Figure 9 - Control of a Resource Subsystem

from the top of the plane to the bottom. However, in order to adjust this influence' strength, the incline plane resource must embody a flexible component which can be adjusted by the controller. If no such component exists, then a new resource which is adjustable by the controller must be injected into the resource subsystem. Frequently, resource subsystem resources are not compatible with the controller, requiring the injection of another resource which is adjustable by the controller and compatible with other resources. An example of such an incompatibility is the coal furnace. The transformational influence (the fire) can not be moderated by any mechanical or electrical controller and is thus incompatible with such a controller. A new resource, such as a damper, must be injected which is compatible with both the controller and existing resources,

and will allow the controller to moderate the transformational influence.

Control enforced upon one or more resources in a resource subsystem will impact the subsystem's structure in two ways.

(1) An increased number of SRs dedicated to the control function. Since control, if present in a resource subsystem, is instrumental to achievement of the design objective, the additional SRs injected to gain control will become an integral part of the resource subsystem's structure. Each control SR must be adjustable by the controller, so it will have multiple possible states. The complexity of the resource subsystem's structure will increase due to the number of multiple-state resources it contains.

(2) Each resource brings with it a number of interrelationships. More resources implicates a greater structural complexity due to a larger number of interrelationships between resources.

INTEGRATING EXAMPLES

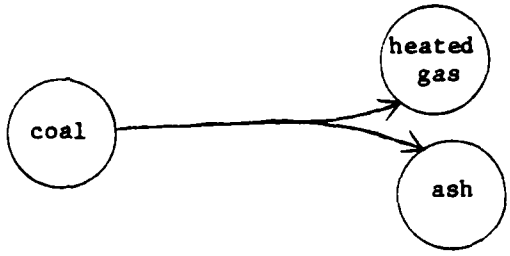
As an example of a single transformation step, consider a lump of coal (figure 10a). If the lump of coal were to be uncontrollably burned [objective of the transformation], it would go from a lump of coal [initial state] to a pile of ashes [terminal state]. Many of the attributes of the lump of coal change value during this transformation: its

color goes from black to white, its rigidity goes from that of a solid mass to that of a white powder, and its heat energy capacity goes from whatever it was to zero. This relatively simple transformation occurs naturally and thus needs no control (figure 10b).

However, had the objective been to extract as much heat energy from the coal as possible, then the rate of natural transformation must be altered. Control becomes necessary. One common way to achieve the desired control objective would be to control the rate of air flowing to the transforming coal. A second resource becomes conceptually involved. The relevant attribute of the resource air is the rate at which it is exposed to the coal. A controller must be introduced to adjust the value of this second resource's attribute [air flow rate] to achieve optimum heat output [objective] of the air and coal system.

On the other hand, suppose the objective had been to keep a door open. The involved resources would be the lump of coal and a door. The relevant attribute for both the coal and the door would be position. The controller must position the door, then the coal so as to achieve the transformation objective. The door has gone from some degree of being open [initial state] to being restrained open [terminal state]. The coal has been transformed from not being in a position to restrain the door [initial state] to being in a position to restrain the door [terminal state].

Note that the concept of a resource attribute



NOTE: availability of air is taken for granted.

Figure 10a - Simple Coal Burn

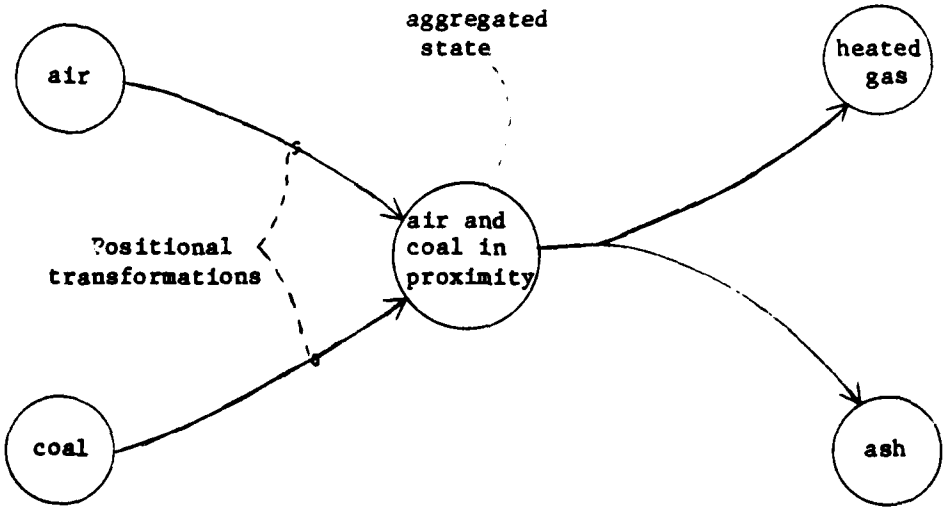


Figure 10b - Simple Coal Burn: more precise

encompasses dynamic constructs, such as the airflow attribute of air. Stability of such dynamic attributes would be reflected by a value which is constant over the term of the transformation. In the burning coal example, the airflow would remain in constant proportion to the quantity of coal undergoing oxidation throughout the transformation until all of the coal (or all of the oxygen) were consumed. The ratio would then change at this point of resource exhaustion, delineating the term of the transformation by a change in the state of both involved resources.

APPLYING TRANSFORMATION ANALYSIS

Transformation analysis is performed without regard to any intended follow-on application. Observe that transformation analysis is a general application technique by which to analyze a set of interacting resources to expose the fundamental functional relationships that underlie the interactions. The resource subsystem of a controlled system is such a set of resources, with the additional uniqueness that its member resources cannot be expected to interact desirablely without the external influence from a controller. When transformation analysis is applied to the desired behavior of a resource subsystem, each fundamental interrelationship which will require controller intervention - i. e., each control need - will be among the interrelationships exposed. The controlled system controller must be designed to accommodate all the control need of the

resource subsystem. In a SCCS, the controller includes software which must be designed to accommodate its apportionment of control needs. In this design criterion lies the utility of transformation analysis to software design. And in this same design criteria - i. e., the relationship between control needs of the resource subsystem and the controls provided by the software - lies the prospective utility of the resource subsystem's structure (as derived through transformation analysis) as a foundation for software design cost projections.

In the following section, the theory presented in this section is summarized into a step by step procedure for conducting transformation analysis. The rationale for continuing this section into the next is explained in the first subsection of the next section.

CHAPTER III

CONCEPT DEMONSTRATION

PERSPECTIVE AND OVERVIEW

In demonstrating the concepts developed in the previous section, a transformation state diagram will be developed to depict the structure of the resource subsystem defined by a sample software development project. The specifications for the sample project, given in the next paragraph, include the specifications for the resource subsystem. Although specifications call for simulation; the resource subsystem will be discussed herein, unless otherwise noted, as an actual resource subsystem without regard to uniquenesses of simulation.

The concepts from the previous section will be summarized into a coherent three-step sequence. In conjunction with each step, that step's application to the development of the state diagram for the sample development project's resource subsystem will be coherently presented as an example. Since each step finalizes the conceptual theory presented in the last section; this section constitutes, in a sense, an extension of the last.

Using the assumption given below and the transformation state diagram developed from the above activity, the author developed the finished software program given in Appendix

A. Appendix B is a narrative description of Appendix A, and includes a description of each relevant module. Only those modules which functionally participated in the transformation process are considered relevant. Relevant modules are explained in more detail in the narrative of Appendix B.

ASSUMPTION I: The structure of a SCCS resource subsystem can be defined adequately for purposes of software development by a transformation state diagram of the resource subsystem, which incorporates only that subsystem's PR(s).

Project Specifications.

The project entails the development of controlling software in the form of a computer Operating System (OS). The various resources of the computer - i.e., the hardware - constitute the resource subsystem of the sample project's SCCS. The project came from a project assigned for Air Force Institute of Technology (AFIT) class EE6.89, Operating System, Spring quarter 1982. The assignment required students to develop software to simulate a multiprogrammed, single processor OS. The class handout which defined the project is shown in Appendix C, but extensive verbal alterations resulted in specifications as delineated in this paragraph. The imaginary computer system upon which the simulated OS is to execute consists of a single CPU, a 500K-byte memory, and a limited Input-Output subsystem. The Input/Output (IO) subsystem contains two discs of

unlimited capacity and a card reader input device. User jobs are simulated by a stream of user job control card images provided by the instructor. These images are to be read by the card reader at some point in absolute time as specified upon the control card image as that job's arrival time. User jobs, simulated by these control images, are to be stored in disc storage until they could be retained in memory prerequisite to CPU execution. Included on the control card image is a priority specification for that job and an indication of the amount of storage (and memory) space it requires. Each use job control card also specifies a number of disc IOs and an amount of CPU time. During the course of accumulating of this specified amount of CPU time, the job is to accomplish the specified number of IO disc accesses and be terminated.

Accomplishing the IOs and the CPU execution time transforms the unprocessed job into a processed job, and the job terminates. When one job times out (has been attached to the CPU for a specific time interval), terminates, or requests an IO operation, the CPU is allocated to another user job.

CONSTRUCTING A STATE DIAGRAM

The following conceptual steps to transformation analysis should aid in constructing a state diagram of the physical transformations within a resource subsystem.

(1) Identify the PRs (there may be only one) and the transformational objectives (there may be only one). The PRs are the resources whose transformation the entire controlled system is designed to assure. They are effectively the purpose of the system's existence. PRs are the inputs into the system, and will generally be consumables. The transformation objectives are "what is desired" (usually, the system's outputs). If a PR transforms cyclicly, then the transformation objective becomes simply to return the PR back to its start state. PRs and objectives form and focus all subsequent analysis. Since the structure of the resource subsystem is defined by the transformations of its PR(s), each PR will require a state diagram of its transformation. The PR's configuration before it enters into transformation constitutes its start state on the state diagram; its configuration as reflected by the design objective constitutes its stop state. If multiple PRs become aggregated together during the course of transformation, their aggregated state should be reflected as a "new" PR on the state diagram (figure 10b). The aggregated state becomes, in a sense, an intermediate "stop" state for each of the aggregate's components.

Insight as to the nature of the transformation can be gained by observing the unit sizes of the PR at its start and stop states. A difference in unit size between the input resource unit and the output unit confirms that aggregation (or disassociation of an aggregate, if the input

unit is an aggregate of the output unit) will be a factor in the transformation. Note that, while a difference in unit sizes confirms aggregation, the equivalency of unit sizes does not confirm the absence of aggregation as a factor in the transformation.

In the sample development project, the design objective is to transform unprocessed user jobs into a processed user job. The only PR is unprocessed user jobs. Since each user job enters the controlled system one at a time, the PR unit size is one single job. PR units enter and complete transformation at the same unit size, reflecting that permanent aggregation can not be conclusively deduced to be a transformational factor. If permanent aggregation does occur, disaggregation must also occur during the course of the transformation. The fact that multiple PR units are concurrently in transformation, though in different configurations at any point in time, is indicative of at least transitory PR aggregates (queues).

(2) Identify the SRs. These are the resources which contribute to the PR's transformation. To assist in discovering all SRs, it may be useful to identify PR attributes which change during the course of its transformation. Envision the values of these relevant attributes at the PR's start-state configuration and at its stop-state configuration. Examine each of the relevant attributes and determine what devices or mechanisms cause their values to

change during the course of the transformation. These devices and mechanisms are SRs.

Temporary aggregates of the PR, such as queues, also constitute SRs. For each instance where a queue of PR forms, determine the queue's medium and object. A queue's medium is any other resource which is essential to the queue's existence as a discernible entity. An example of a queue's medium is the floor upon which a line of people stand. The line (queue) could not discernibly exist without a continuance of floor to support the people (units which are temporarily aggregated) amassed into the queue. A queue's object is the resource whose service the queue is awaiting, such as bank teller for the queue of people. A queue, its media, and its object are also SRs in the transformation of the PR.

In our development project, the SRs which directly act on the unprocessed user job to cause its transformation into a processed job are the CPU and the IO controller, to include the IO disc. The design objective defines that user jobs, as a prerequisite to having completed transformation, must be executed and must perform IO. The CPU and IO controller provide the energies that cause this defined-by-objective transformation to occur. Other SRs contribute less directly to the transformation. These SRs would be the input device, the IO disc, and CPU memory. The card input device transforms the user job's physical form (an attribute) into an electrical form. The IO disc

and CPU memory contribute indirectly in that they provide a path to the job's execution and IO completion. Note that the IO disc participates both directly and indirectly in the job's overall transformation.

Job queues constitute SRs. PR units will be subject to queuing in the input device, the IO disc, and CPU memory. These are the queuing media. They are subject to queuing while awaiting the input services of the card-reading sensor, storage on the IO disc enroute to memory, storage in CPU memory, and execution by the CPU. These are the queuing objects.

In summary, the SRs discovered in the project are the card reader, the IO disc, CPU memory, the CPU, and the IO controller.

(3) Sequence the SRs. Graphically arrange the SRs into the sequence in which they are actually encountered by transforming PR units. Introduce to-from arrows onto the graphic depiction to reflect this sequencing of the transformation steps. The transformation steps are bounded by the transformation states of the PR. The various states which bound the arrows generally reflect the points at which a different SR begins its interface with the PR. Append start and stop state indicators into their appropriate positions within the sequence. As the coalescing activity of this third step may expose incompletenesses in the identification of SRs (step 2) and perhaps even PRs (step 1),

it may prove advantageous to iterate the thought process between all three steps. Figure 11 shows the results of this step for the sample development project.

From the examples given in the above transformational analysis steps, the transformation state diagram of figure 8 was assembled. Figure 8 was used by the author as an aid in developing the software given in Appendix A. The next section examines the correlation between the state diagram of figure 8 and the software given in Appendix A.

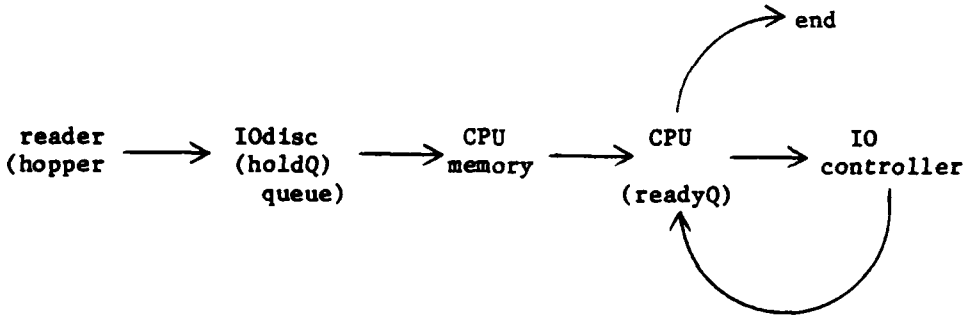


Figure 11 - Sequenced SRs

CHAPTER IV

CONCEPT VALIDATION

PREFACE

This section compares the state diagram developed in the prior section to the software which was generated from it in Appendix A. As specified in Appendix B, only those software modules which participate in the simulation are considered relevant to this thesis. Reference Appendix B for a list of relevant modules.

Before discussion, let us define the following:

transformation step function: A transformation step function can be defined as the following relationship between the immediately following four entities:

(a) one arbitrary transformation step of one arbitrary PR.

(b) all other resources which contribute to (a).

(c) the data images of the arbitrary PR and (b) at the point (a) begins.

(d) the data images of the arbitrary PR and (b) at the point (a) terminates.

A transformation step function is a set of software-performable manipulations upon (c) which maps (c) into (d). One transformation step function exists for each transformation step depicted on a transformation state diagram.

Let us restate the previously posed assumptions and hypotheses - in terms more consistent with the prior two sections, where appropriate. Any further reference in this thesis to a resource subsystem constitutes an implicit reference to the resource subsystem of a SCCS.

ASSUMPTION I: The structure of a SCCS resource subsystem can be defined adequately for purposes of software development by a transformation state diagram of the resource subsystem, which incorporates only that subsystem's PRs.

HYPOTHESIS I: The structure of the resource subsystem is sufficiently predictive of the controlling software's modular structure to be used as a foundation for an as yet undeveloped body of software costing and design management tools and techniques.

HYPOTHESIS IIa (in terms of a general controlled system): There exists a deterministic relationship between the control needs (page 41) of a resource subsystem and the control actions taken by the controller to control the resource subsystem.

HYPOTHESIS IIb (in terms of the software of an SCCS): There exists a deterministic relationship between the control needs of a resource subsystem and the software-units comprising the controller software.

If hypotheses I and IIb are true, then the deterministic

relationship might well suffice as a foundation, itself based upon the structure of a resource subsystem, upon which to develop a set of software development costing tools and techniques.

For purposes of initiating an exploration into the prospective truths of hypotheses I and IIb, let us restrict our scope in the following four ways:

(1) We shall take a software module to be a software-unit. This is consistent with the definitions of both concepts, and may disclose a convenient association between a readily discernible coding aggregate (a module) and a resource subsystem's control needs.

(2) The transformation step functions of a transformation state diagram embodies a projection of the fundamental interrelationships between the interacting resources within the resource subsystem, to include control needs. Since our sample project is a simulation, all the transformations are to be simulated by - i. e., accomplished within - the control software. Thus, all of the transformations of the transformation state diagram represent a control need.

(3) As a result of this deterministic relationship, the control needs can be said to "predict" the presence of a software-unit in the control software if the software-unit's presence is exposed by virtue of the deterministic relationship between a the resource subsystem structure (a transformation state diagram, according to [2] above) and

software-units (modules, according to [1] above).

(4) The deterministic relationship is projected in a one-to-one correlation between control needs and software-units. It is recognized that other prospective relationships may well exist other than one-to-one correlation. However, one-to-one correlation is logically convenient to conceptualize, and will suffice as a point of beginning for this preliminary exploration. One-to-one correspondence would reflect the philosophy that one software-unit could be designed to accommodate precisely one entire transformation step. An exploration of other relationships will be deferred to further research.

ONE-TO-ONE TEST

These constraints permit us to form hypothesis III:

HYPOTHESIS III (for test purposes): There exists a one-to-one correspondence between the transformation step functions of a state diagram and the software modules comprising the control subsystem software of the corresponding SCCS.

Restrictions (1), (2) and (3) taken together also project the following:

HYPOTHESIS IV (for test purposes): the software-unit whose presence is predicted by a transformation state diagram is a module.

By comparing a count of relevant software modules given in Appendix B with a count of the number of transformation steps from figure 8, hypothesis III is trivially disproven. By observation, there are 12 transformation step functions projected in the state diagram, while there are 22 modules in the software. A one-to-one relationship is not possible unless the number of software modules equals the number of transformation step functions that exists in the state diagram from which the software is generated.

TEST OF NO RELATIONSHIP

To further explore the precise nature of the relationship between the structure of the resource subsystem (as projected in a state diagram) and the modular structure of software (in simulation-type control software), let us continue by posing the following hypotheses:

HYPOTHESIS V (for test purposes): There exists no relationship between the transformation step functions of a state diagram and the software modules comprising the control subsystem software of the corresponding SCCS.

For Hypothesis V to be true, the number of one-to-one relationships between transformation step functions and software modules must be either zero or sufficiently small so as to be discarded as coincidental. Therefore,

HYPOTHESIS VI (for test purposes): There are no occurrences

of one-to-one correspondence between the transformation step functions of a state diagram and the software modules comprising the control subsystem software of the corresponding SCCS.

In prelude to discussion, let us define the following four concepts:

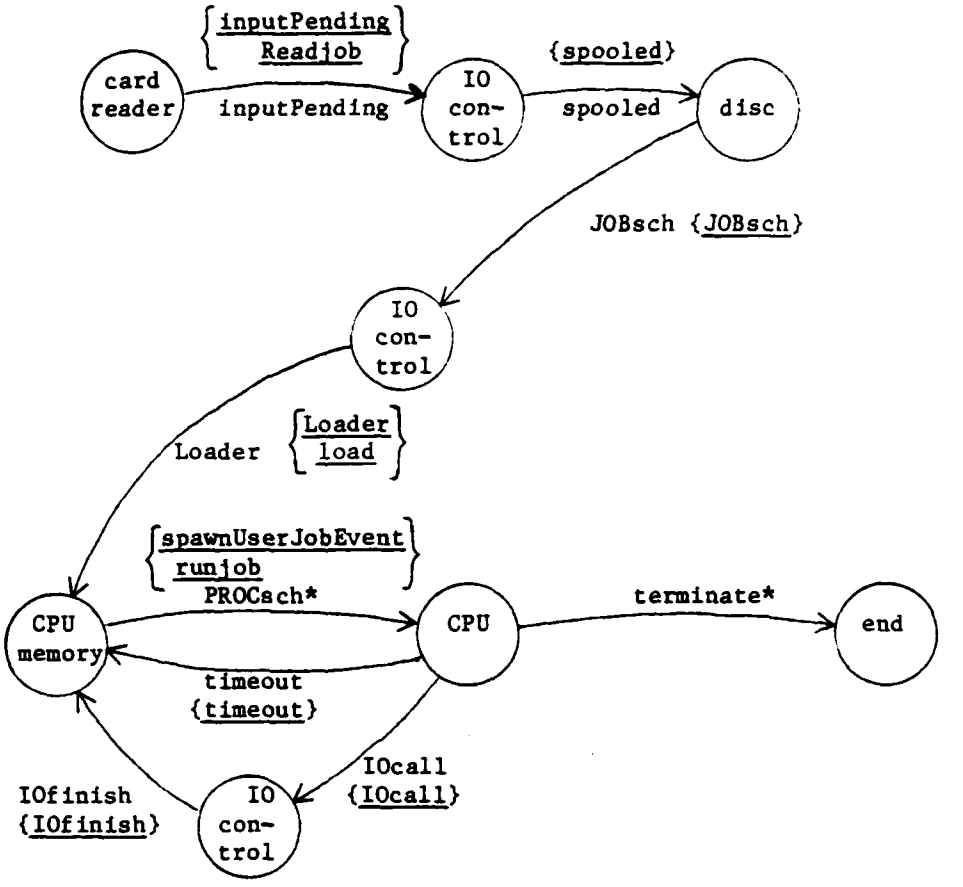
(1) primary module: if the whole or partial modular function of a software module can be uniquely associated with one and only one transformation step function, then that software module is a primary module. These modules are shown in figure 12 overlaid against the transformation step function to which the correlate.

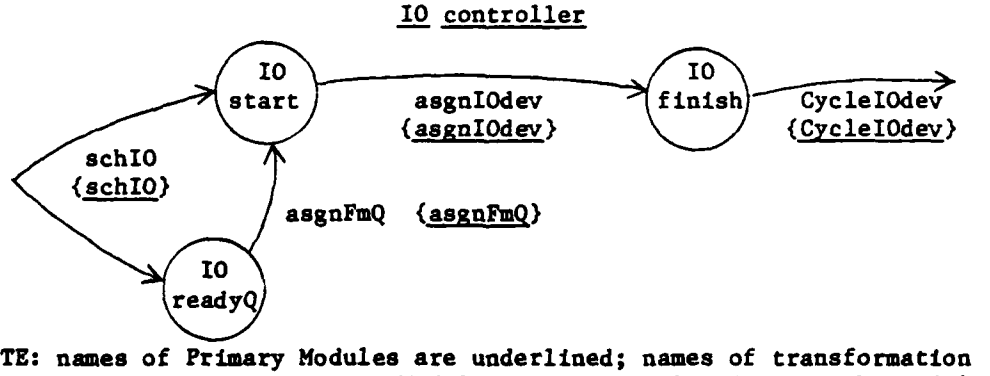
(2) primary module set: the set of one or more primary modules associated with one transformation step function. Also see figure 12.

(3) one-to-one correspondence (between a transformation step function and a primary module set): a transformation step function and a primary module set exhibit one-to-one correspondence if and only if the following two conditions are met:

(a) execution of the code comprising the primary module set accomplishes one whole completion of its associated transformation step function; and

(b) execution of the same code could not possibly further the completion of (i. e., could not either partially or totally complete) any other transformation step





NOTE: names of Primary Modules are underlined; names of transformation steps are not. Primary Module sets are enclosed in brackets { }. *PROCsch contributes to both transformations, but is not a Prime module.

Figure 12 - Primary Modules and Primary Module Sets

function.

(4) optimization consideration: a focus by software developers upon the performance of his software code as measured upon the physical components with which it will interface. Examples of optimization considerations are a developer's focus upon minimizing his code's execution time , minimizing his code's requirements for memory, and maximizing his code's understandability to enhance human interface factors. These considerations frequently conflict with each other between design alternatives, requiring the software developer to select a design alternative which optimizes these factors for a specific situation.

Twelve primary module sets are depicted in figure 12. Of these twelve primary module sets, eleven are one-to-one correlated to a unique transformation step. The set containing spawnUserJobEvent (note that module names are underlined, in addition to definitions. Transformation step names are not underlined) and runjob does not accomplish the entire transformation step function, since part of the transformation step function performed by the module PROCsch is missing. (Since PROCsch embodies two transformation step functions, it is not a primary module, and hence cannot be included in a primary module set; see case 7 below.) The set containing inputPending is one-to-one correlated, though it may superficially appear not to be. Readjob appears to fail the definitional criteria of a pri-

mary module. It is a called by inputPending and by the main loop initialization module. Though it is a part of two modules, it participates in only one transformation step function. Hence, Readjob is a primary module; and, as a set member with inputPending, the set is one-to-one correlated with a transformation step function. (Case 4 below discusses this complication in additional detail.) All other primary module sets are one-to-one correlated with their corresponding transformation step function.

Let us now define one-to-one correlation (between a transformation step function and a primary module [not a primary module set]) to exist if and only if the primary module is a single member of a primary module set which was one-to-one correlated with a transformation step function in the last paragraph. Since there were eleven one-to-one correlated primary module sets, there are eleven prospective one-to-one correlated modules. However, the primary module sets formed by inputPending and Readjob (case 4 below), and by load and Loader (case 6 below), are not single-member sets. The primary modules of these two sets are hence not one-to-one correlated primary modules (although they remain one-to-one correlated as primary module sets). Hence, nine modules are one-to-one correlated (spooled, JOBsch, Timeout, IOcall, IOfinish, schIO, asgnIOdev, asgnFmQ, and cycleIO). With nine of the 22 simulated OS modules one-to-one correlated with a transformation step function from the state diagram, the prospect

of a coincidental cause to the correlation can be comfortably discounted. Hypotheses V and VI are thus disproven.

OBSERVATION

The one-to-one correspondence between primary module sets and the transformation step functions could be improved with relatively minor adjustments to the modular structure of the software. If some degree of duplication were acceptable (see case 7 below), then the termination software-unit could be extracted from PROCsch, forming a residual "PROCsch" module and a "termination" module (quotes ["] highlight the newly formed hypothetical modules). The hypothetical "termination" module would meet the definitional criteria for a primary module, as would the residual "PROCsch" module. The "termination" module meets the definitional criteria for one-to-one (primary module set) correspondence with the termination transformation step function; while the "PROCsch" module will join with runjob and spawnUserJobEvent to form a one-to-one correspondence with the PROCsch transformation step function. With such a dissociation of software-units from PROCsch, the one-to-one correspondence between primary module sets and transformation step functions would become 100%.

At least one other design alternative is also conceivable to accomplish this same functional dissociation of PROCsch's functions. The job control block could be

modified to include space for a second pointer to the preceding user job in the queue. This design strategy would eliminate the need to scan readyQ in order to acquire the location of the prior job control block, thus permitting the termination function to be performed without a need to scan readyQ. This design strategy would also permit the separation of the termination software-unit from PROCsch. However, implementation of this strategy over the last one would also require the sacrifice of certain possible coding optimizations. Memory space must be provided for an additional pointer value in each job's control block. Another PR attribute would be introduced, which would require additional execution time to update, as well as a slight volume of additional coding to accomplish the updating.

The point to be raised from these observations is not that a perfect correlation between primary module sets and transformation step functions is achievable. Instead, the important observation is that all three alternatives - the alternative chosen by the author and the two different design alternatives discussed above - performed virtually identical functions, but each posed different optimization alternatives. Selection of alternatives was made not based on function, but on optimization considerations.

ANALYTICAL CASE STUDIES

With hypotheses III and VI disproven, it appears that a

non-deterministic relationship exists in a SCCS between the structure of the resource subsystem (as projected in a state diagram) and the modular structure of control subsystem software. At least, this conclusion seems pertinent to the simulated OS subject of this thesis. Hypothesizing without further analysis about the precise nature of such a non-deterministic relationship would be largely a matter of speculation. Hence, the remainder of this section undertakes a detailed analysis of the design alternatives facing the software's author during software design. The analysis focuses on which alternative the author selected, and more importantly why. Of importance are those cases where the author selected design alternatives founded on criteria other than adherence to the transformation step functions of the state diagram. The intent of this undertaking is to expose more details of the nature of the non-deterministic relationship exposed above.

Let us recall the definition of "software-unit" from our accumulated glossary (Reference page 14).

The following cases address several specific modules and modular functions. It is not intended that all modules be discussed. It is intended that as many as possible justifications be revealed for deviating from state-diagram-based design criteria. Refer to Appendix B for a functional description of the modules mentioned.

(1) fileXSevent would not be present if the "dispose" function of the PASCAL compiler had been functioning

properly.

(2) fileEventIO and filejob are virtually identical functions. Optimizations in memory requirements should be readily achievable by combining these two utility modules into one module. However, PASCAL syntax rules preclude these modules from being joined into one module.

(3) The software realizes optimizations in memory requirements by incorporating the searching and filing overhead functions of both eventQ and IOreadyQ into the one module fileEventIO. Achievement of the memory optimization did not precipitate a need for any additional modules; but, it did precipitate changes in the contents of other IO processing modules and in the design of a common event and IO control block.

(4) When the simulated OS is operating in steady state, the module Readjob is a subroutine of inputPending. However, the functions performed by Readjob are also required for initialization of eventsim. Thus, Readjob must be extracted from inputPending into a separate module in order to avoid extensive code duplication for a one-time use of the Readjob function. Readability and memory use are optimization drivers for the physical separation of two functions into two modules.

(5) spawnUserJobEvent is referenced only by runjob. It should have been embodied within runjob as a subroutine, and not separated as an external module.

(6) load is called only from Loader, and is thus a

proper subroutine thereof. However, PASCAL syntactical constraints preclude a subroutine in module within a "case" statement.

(7) A termination function is projected by the state diagram as a discernible function. It is indeed a discernible function within the software, but is not isolated as a separate module. The termination transformation step function is performed by the software-unit comprised by lines 388-394 of PROCsch (see Appendix A). The remainder of PROCsch performs the PROCsch transformation step function (figure 12). Since PROCsch performs multiple transformation step function, it does not meet the definitional requirements of a primary module.

The reason why the termination function is embedded within another module is two-fold. First, the design of readyQ did not include a pointer to the preceding job control block. Hence, the entire queue had to be scanned in order to obtain the necessary pointers to remove a control block from readyQ. Thus, commitments to design alternatives earlier in the design project contributed to the incorporation of one transformation step function into a module which would otherwise be a primary module of a separate transformation step function. However, this earlier design alternative was selected to reduce OS's memory requirements by not requiring the additional storage space for the second pointer. In the end, this first reason comes down to optimization considerations.

Second, PROCsch performs a readyQ scan during its search for the next user job to execute. Since incorporation of the termination software-unit into PROCsch would eliminate a need to duplicate PROCsch's scanning software-unit into another module, the optimization driver of reducing memory usage again contributed to the incorporation.

(8) Optimization led to the partitioning of the PROCsch software-unit into the two modules PROCsch and runjob. PROCsch scans readyQ to determine the next job to be executed. If there were no other jobs ready for execution, PROCsch will discover that no jobs are ready only after it scans the entire readyQ. Whenever a user job timed out or completed IO activity, that same user job would have to be reexecuted without an intervening job if there were no other user jobs ready for execution. If PROCsch were again called to reexecute the same user job, it would scan readyQ a second time. By separating the software-unit that scans and the software-unit that executes into two modules, the execution time involved with the superfluous scan can be eliminated.

(9) Commitment to the design strategies given in cases 7 and 8 above precipitates a need for an additional module. Jobs which just timed out or just completed IO may have completed their execution. If so, the job must be routed for termination in lieu of reexecution. assignjob performs this routing function. Since cases 7 and 8 were precipitated by optimization considerations, the addition

of assignjob must also be attributed to optimization considerations.

(10) getEventIOblk, fileEventIO, fillEventIOblk, fileXSevent, and filejob, assignjob and PROCsch do not appear in the primary module sets of figure 12. Case 7 and 9 above detail why PROCsch and assignjob, respectively, are not primary modules. The other modules listed pertain to the transformations of SRs. getEventIOblk and fillEventIOblk accomplish functions pertinent to a control block, while fileEventIO pertains to the SR IOreadyQ, and filejob pertains to the SRs jobQ and readyQ. fileXSevent is discussed in case 1 above.

While the author makes no pretense that the above list is complete, it does highlight several prominent bases other than the state diagram for the selection of design alternatives. In summary, the following bases emerge to preempt adherence to a state diagram design basis:

(1) Optimization considerations precipitated the most profound volume of preemptions (cases 2, 3, 4, 7, 8 and 9). Cases 3 and 7 reduced the module count, while cases 4, 8 and 9 increased the module count.

(2) Syntax constraints of the coding language precipitated a need to deviate from state diagram structure (cases 2 and 6). Syntax considerations either precipitated additional modules beyond the projection of the state diagram (case 6) or precluded the realization of possible optimizations (case 2).

(3) Disfunctional support tool (case 1) and programmer error (case 5) both resulted in additional modules being spawned.

(4) Four modules were required above those projected in the state diagram to accommodate SRs (case 10).

CHAPTER V

SUMMARY AND CONCLUSIONS

SPECIFIC CONCLUSIONS

Let us preface this section by defining the following two entities:

"A" refers to the modular structure of the software program as developed to control an arbitrary resource subsystem whose desired behavior is described in the specifications of a SCCS. The structure for the software developed as a sample in this thesis (Appendix A) is an example of "A".

"B" refers to the structure of the same resource subsystem as derived through the transformation analysis process described in this thesis, and depicted in a resulting transformation state diagram. The structure of the resource subsystem developed as a sample in this thesis (figure 8) is an example of "B".

The mismatched count of fundamental structural elements between A and B provides conclusive evidence that a one-to-one correspondence between A and B does not exist (test hypothesis III is false). However, the number of occurrences of one-to-one correspondence which appeared in the sample comparison suggests that a relationship of some as yet unknown nature does exist between A and B (test

hypotheses V and VI are false). In terms of the restrictions imposed earlier (page 53), B seems to have some as yet unclear utility as a predictor of A.

Evidence implicated that the software-units whose presence in the software is best "predicted" by the transformation state diagram is not software modules (test hypothesis IV is contradicted). The transformation state diagram projected the presence of thirteen software-units. Primary module sets are software-units that are not modules. They are by definition comprised of one or more (primary) software modules. Eleven primary module sets exhibited a one-to-one correspondence with eleven of the thirteen transformation step functions. Primary modules are software-units that are modules. Nine primary modules exhibited one-to-one correspondence with nine of the thirteen transformation step functions. Hence, the transformation state diagram was a better predictor of primary module sets than of primary modules. Although this evidence does not contradict hypothesis I, it is suggestive that the structure of the resource subsystem may be a better predictor of some unit of software other that modules.

Evidence does not, however, implicate the transformation state diagram to be a good predictor of the totality of a program's modular structure. The software for the sample project embodied 22 relevant modules, whereas the transformation state diagram projected only thirteen. This is not to say that the modularization of the software could

not have been designed to subscribe precisely to the thirteen transformation step functions projected in the transformation state diagram. Let us presume that the software could be modularized precisely as projected by the transformation state diagram - i. e., into thirteen modules whose functions correlated precisely with the functions projected by the transformation state diagram. Then the functions of 22 minus 13, or nine of the non-primary modules, would have to be absorbed into the 13 primary modules whose functions correlate with the transformation state diagram. Since the functions of many of the non-primary modules are used by several other modules, duplication would be unavoidable. This duplication would probably be undesirable from the standpoint of code optimization considerations.

Optimization Considerations.

The observation from the prior section constitutes further evidence that the one-to-one correspondence between primary module sets and transformation step functions could be improved if certain coding optimization concessions were made during software design. By splitting the module PROCsch into the termination module and a residual PROCsch module, the primary module set formed by runjob, spawnUserJobEvent and the PROCsch residual would correlate one-to-one with the PROCsch transformation step function, while the termination module would one-to-one correlate

with the terminate transformation step function. There would then exist thirteen one-to-one correlated primary module sets for thirteen transformation step functions. The modularization design alternatives facing the software designer were functionally equivalent, so the criteria which precipitated selection could not have been based on function. The particular design alternative selected was based on optimization criteria.

The fact that the designer relied on optimization criteria to resolve his modularization structure contradicts hypotheses I and II, and thus also hypotheses III and IV which are based on hypotheses I and II. Information pertinent to the selection among modularization alternatives appears to have another basis - optimization criteria - in addition to that of function. The modularization alternatives considered were functionally equivalent: the same transformation step functions were equally appropriate to each alternative. Since the alternatives were functionally equivalent, the function of the alternatives did not reflect the differences in optimization criteria that precipitated the selection of the chosen modularization alternative over the others. It appears that there may exist (at least) two independent determinants to the selection of software modularization design alternatives.

If two independent determinants do exist, then attempting to form a deterministic relationship from just one of the independent determinants - as proposed in hypothesis II

- would ultimately prove unsuccessful. Hypothesis II is contradicted in that a deterministic relationship cannot exist between software modular structure and only one of two independent determinants. Hypothesis I is contradicted in that the structure of a resource subsystem cannot by itself be "sufficiently predictive" of the controlling software's modular structure. The contribution of the other independent determinant - optimization criteria - must be incorporated into any prospective deterministic relationship. A deterministic relationship may be deducible reguarding the modularization of software, but such a relationship must incorporate all independent determinants. Formal proof of independence between the two prospectively independent determinants of design selections is deferred to further statistical research. Prerequisite to achieving such a deterministic relationship, a measurement vehicle by which to measure optimization considerations must be devised. Until such a vehicle can be devised, any relationship will continue to be stochastic - not deterministic - between modular structure and any one of the independent determinants in the absence of others. Development of any such vehicle is also deferred to later research.

GENERAL CONCLUSIONS

Let us presume that these two determinants of software modularity - the functional determinant and the optimization determinant - are independent. One can then expect

that software which is developed rigidly to the dictates of the functional determinant alone will lack important elements of code optimization.

Transformation Analysis in Software Design.

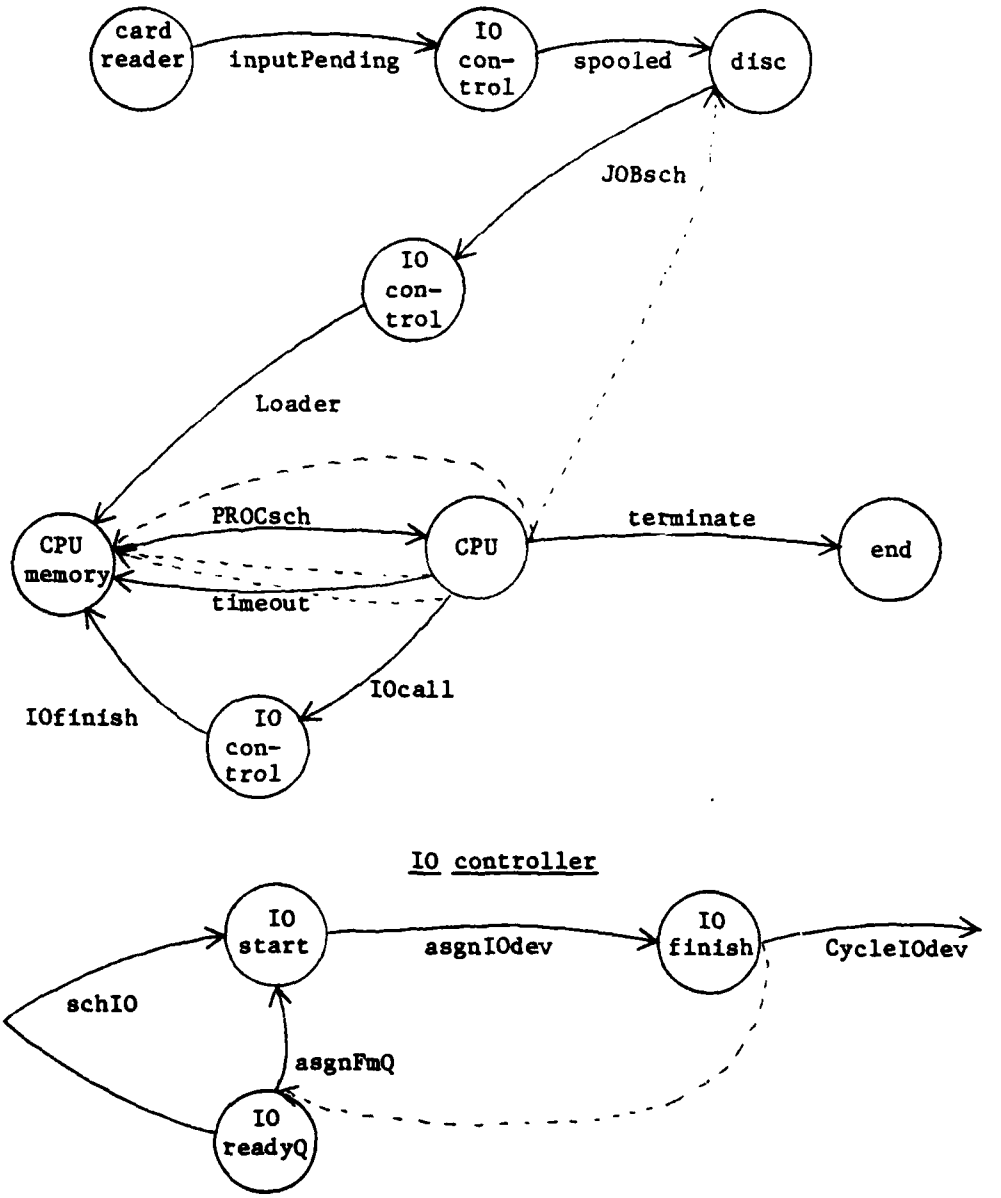
Assumption I was premature. Four modules were present in the software whose functions where prominently of a SR (see case 10 of prior section). Though the structure of a resource subsystem has been defined as the sequence of transformations through which the PR(s) progress, a more accurate reflection of modular count and structure might be achieved for software design applications by incorporating the SRs into the structural depiction.

Aside from the managerial focus of this thesis, the author concluded that transformational analysis and the resulting transformation state diagram proved to be useful and effective design tools. Transformation analysis provided an excellent perspective of the project's entire system - a perspective of both controller software and resource subsystem behavior. The resulting transformation state diagram is sufficiently indicative of the structure of software as to constitute a starting point for detailed software design.

During the course of software design, the author observed a major informational deficiency in the transformation state diagram he used. The deficient information was generally derived during the course of step three of the

transformation analysis procedure, but was not carried over onto the transformation state diagram. Frequently, the start or finish of one transformation step for one PR unit was the impetus for the start of a different transformation step for a different PR unit. For example, when the termination transformation step of one user job concluded, the JOBsch transformation step of another job began. Depiction of this impetus-type information on the transformation state diagram would be helpful during software design activities. A transformation state diagram format incorporating this improvement is depicted in figure 13.

The author suspects that the use of a transformation state diagram to design software other than simulation software might expose an additional informational deficiency in the transformation state diagram. In the sample simulation software design project, all resources and transformations are simulated. All transformations present in the transformation state diagram become functions to be performed by the software. However, in a non-simulated SCCS, some of the resource subsystem transformations will and others will not fall under software control. While the transformation state diagram does expose the fundamental transformations of a resource subsystem, it does not seem to clearly project which of the transformation steps will be subject to interface with the controller. Only those transformation steps which interface with the controller will require representation within the software control



NOTE: Impetuses are shown as dashed arrows.

Figure 13 - Transformation State Diagram with Impetuses

program. Though the details of applying transformation analysis to a real SCCS appear interesting, this application must be deferred to further research.

APPENDIX A

SOFTWARE CODE

```
 1 program sim (JobsIn(LFN = 101),  TAdistr(LFN = 102,
 2   LINELENGTH=65), memDistr(LFN=103,LINELENGTH=133),
 3 IOdistr(LFN=104,LINELENGTH=133), output(LFN=105,
 4 LINELENGTH=133));
 5
 6 label
 7   9999; {program end, in event of premature termination}
 8
 9 const
10   timesup = 0.5;          {time slice allowed before timeout}
11   readlimit = 200;        {limit of number of input jobs to accept}
12   IOdevs = 2;             {number of disks in system}
13   sysDev = 2;             {disk containing the system files}
14   memsize = 500;          {size of memory}
15   IOchkIntvl = 180;       {interval, in sec, over which IO count is
16                            taken; < 75 minutes (360 sec = 6 min)}
17
18 type
19   jobptr = ^jobBlock;
20   eventIOptr = ^eventIOblock;
21   state = (hold, ready, exec, IO);
22   Events = (timeout, IOcall, IOfinish,  {ref cases in 'eventsim'}
23             spooled, Loader, inputPending, IOtally);
24   filetype = (mem, IO2);  {types of output files}
25   jobBlock = record
26     arrive: real;         {absoulte time of job's arrival at reader}
27     name: packed array [1..4] of char;  {jobname}
28     CPU: integer;         {initial amount of CPU time requested}
29     CPUrem: real;         {CPU time left till completion}
30     mem: integer;         {memory requested by job}
31     priority: integer;    {priority assigned to job}
32     IOrem: integer;       {# of IO requests remaining}
33     cumIOtime: real;      {cumulative IO time}
34     index: real;          {priority by which job is queued & run}
35     activity: state;      {transient state of job}
36     time2IO: real;        {time interval till next IO}
37     IOintvl: real;        {time between IO requests}
38     nextNQ: jobptr        {pointer to next job in jobQ or readyQ}
39   end;  {jobBlock record}
40   eventIOblock = record
41     eventType: Events;
42     index: real;         {index by which block is filed on eventQ or
43                           IOreadyQ and processed.  For eventQ, it
44                           is absolute time; for IOreadyQ it = index}
45     device: integer;      {device assigned to an IO event.  Record
46                     of device must be retained in event block in an
47                     lieu of job block, which system jobs dont have}
48     job: jobptr;          {job associated with event}
49     nextNQ: eventIOptr    {pointer to next event in queue}
50   end;  {eventIOblock record}
51
```

```
52 var
53   JobsIn: text;              {input file}
54   TAdistr: text;             {output for turn-around distributions}
55   memDistr: text;            {output for memory distributions}
56   IOdistr: text;             {output for IO distributions}
57   system: set of Events;     {Events which the system does}
58   whichone: integer;         {counter through IO time simulator}
59   clock: real;               {system absolute clock time}
60   memrem: integer;           {memory remaining (awaiting allocation)}
61   readcount: integer;        {count number of jobs read}
62   x: integer;                {local counter for loops}
63   LoaderAvailable: boolean;  {precludes superfluous Loader activity}
64   IOdev: array [1..IOdevs] of eventIOptr;  {pointer to control block
65                                             associated with each device}
66   eventQ, XSeventIOq: eventIOptr;  {pointers to queue heads}
67   IOreadyQ, sysIOq: eventIOptr;  {pointers to queue heads}
68   jobQ, readyQ, XSjobQ: jobptr;  {pointers to queue heads}
69   event: eventIOptr;         {pointer to current event being processed}
70   run: jobptr;               {pointer to executing job}
71   IOtime: array [1..15] of real;  {pseudo-random IO time intervals}
72
73 {STATISTICAL VARIABLES}
74
75   TA, WTA, WTApr: real;    {turn-around, weighted TA; ref specific
76                              computations for differences}
77   jobcount: array [1..5] of integer;  {counter for priority}
78   cumTA: array [1..5] of real;  {cumulative TA by priority}
79   cumWTA: array [1..5] of real;  {cumulative WTA by priority}
80   cumCPUIO: array [1..5] of real;  {cum run & IO times by priority}
81   maxTA: array [1..5] of real;
82   maxWTA: array [1..5] of real;
83   IOcount: integer;          {# of IOs in period}
84   field: array [filetype] of integer;  {output page line}
85   time: array [filetype, 1..11] of real;  {time element of field}
86   count: array [filetype, 1..11] of integer;  {count within field}
87   sysIO: real;               {accumulates system IO time}
88  {to compute weighted average of memory use}
89   startTime: real;           {remember the first start time}
90   memhold: integer;          {remember the last memory value}
91   memtimehold: real;         {remember the last memory statistic time}
92   weightedmem: real;         {numerator in weighing equasion}
93   avgmemuse: real;           {final computation}
94 {*****************************************************************}
95 {--------------EVENT (Interrupt) PROCESSOR----------------------}
96 procedure eventsim;
97
98 label 999,                   {after last input job has been read}
99       990;                   {loop control in 'inputPending'}
100
101 var
102   hold1, hold2: jobptr;      {for temporary local retention}
103   nextjobin: jobptr;         {pointer to job awaiting arrival time}
104   x: integer;                {loop counter}
```

```
105 {-----------STATISTICAL ACCUMULATIONS-------------}
106 procedure fileEventIO (Qpos: eventIOptr; var job4: eventIOptr);
107 {*********************}                                    forward;
108 procedure format (var outfile: text; typex: filetype;
109                   {cont'd} var kount: integer; var eventTime: real);
110   {formats output so as to print across the output page}
111 var x: integer;
112 begin
113 count[typex, field[typex]]:= kount;
114 if typex = mem then       {accumulate numerator in mem weighted avg}
115   if memtimehold = eventTime  {use min 'memrem' for single time}
116     then if kount < memhold then memhold:= kount else {dummy else}
117     else begin
118       weightedmem:= weightedmem + ((eventTime - memtimehold) *
119                                    memhold);
120       memhold:= kount;    {remember this value for next cycle}
121       memtimehold:= eventTime  {"}
122     end;  {if memtimehold/else}
123 if eventQ^.nextNQ = nil   {is this the final event?}
124   then begin              {yes it is}
125     time[IO2, field[IO2]]:= XSeventIOq^.index/3600.0; {prior ev end}
126     for x:= 1 to field[IO2] do
127       write (IOdistr, time[IO2, x]:6:2, ':', count[IO2, x]:5);
128     writeln (IOdistr);
129     for x:= 1 to (field[mem] - 1) do
130       write (memDistr, time[mem, x]:6:2, ':', count[mem, x]:5);
131     writeln (memDistr);
132     avgmemuse:= weightedmem / (eventTime - startTime)
133   end  {if/then}
134   else begin              {no, its not the final event}
135     time[typex, field[typex]]:= eventTime/3600.0;
136     if field[typex] < 11  {11 fields across 132-char page}
137       then field[typex]:= field[typex] + 1 {increment line position}
138       else begin          {completed line ready to be outputed}
139         for x:= 1 to 11 do
140           write(outfile,time[typex, x]:6:2, ':', count[typex,  ]:5);
141         writeln (outfile);
142         field[typex]:= 1  {re-set line position indicator}
143       end;  {if field/else}
144     if typex = IO2 then begin  {additional processing for IOdistr}
145       kount:= 0;          {reset counter}
146       event^.index:= event^.index + IOchkIntvl; {next chk-point}
147       fileEventIO (eventQ, event)  {file next chk-point}
148     end  {if typex/then}
149   end  {if eventQ/else}
150 end;  {procedure format}
151 {***********************}
```

```
152 procedure turnaround (jobz: jobptr);
153   {called upon job termination to record its turnaround statistics}
154 var CPUreal: real;          {to expedite arithmetic object code}
155 begin
156 format (memDistr, mem, memrem, clock);  {record memory change}
157 with jobz^ do begin
158   CPUreal:= CPU;            {convert to real}
159   TA:= clock - arrive;
160   WTA:= TA / (CPUreal + cumIOtime);
161   writeln (TAdistr, ' ', name:4, priority:2, arrive/3600.0:9:4,
162      clock/3600.0:9:4, CPU:6, cumIOtime:10:5, TA:13:6, WTA:11:6);
163   cumCPUIO[priority]:= cumCPUIO[priority] + CPUreal + cumIOtime;
164   cumTA[priority]:= cumTA[priority] + TA;
165   cumWTA[priority]:= cumWTA[priority] + WTA;
166   if TA > maxTA[priority] then maxTA[priority]:= TA;
167   if WTA > maxWTA[priority] then maxWTA[priority]:= WTA;
168   jobcount[priority]:= jobcount[priority] + 1
169 end  {with}
170 end;  {turnaround}
171 {---------IO & EVENT BLOCK GENERAL PROCESSING UTILITIES----------}
172 procedure getEventIOblk (var loc1: eventIOptr);
173   {returns to caller a block for use as an event or IO control blk}
174 begin
175 if XSeventIOq = nil
176   then new (loc1)
177   else begin
178     loc1:= XSeventIOq;
179     XSeventIOq:= XSeventIOq^.nextNQ
180   end;  {if/else}
181 loc1^.job:= nil;            {set selected fields of control blk}
182 loc1^.nextNQ:= nil
183 end;  {getEventIOblk}
184 {***************************}
185 procedure fileEventIO;
186   {files 'job4' onto the queue 'Qpos' in order of index value;
187    lower index values are closer to the queue head}
188 var fileafter: eventIOptr;  {pointer to block after which job4 goes}
189
190 procedure scan (loc9: eventIOptr);  {recursive subroutine}
191   {follows chain to the correct location for 'job4'}
192 begin
193 fileafter:= loc9;
194 if loc9^.nextNQ <> nil
195   then if job4^.index >= loc9^.nextNQ^.index
196     then scan (loc9^.nextNQ)
197 end;  {scan}
198
199 begin {fileEventIO}
200 scan (Qpos);
201 job4^.nextNQ:= fileafter^.nextNQ;  {insert 'job4' into queue}
202 fileafter^.nextNQ:= job4
203 end;  {fileEventIO}
204 {***************************}
```

```
205 procedure fillEventIOblk(ndx:real;typex:Events;var loc2:eventIOptr);
206    {fills non-control fields within the event or IO cntl blk}
207 begin
208 with loc2^ do begin
209   eventType:= typex;
210   index:= ndx
211 end end;  {with & procedure fillEventIOblk}
212 {*****************************}
213 procedure fileXSevent (var loc3: eventIOptr);
214   {accomodates the non-functional "dispose" intrinsic function.
215    expended control blocks are retained in a queue for later reuse}
216 begin
217 loc3^.nextNQ:= XSeventIOq;
218 XSeventIOq:= loc3
219 end; {fileXSevent}
220 {*****************************}
221 procedure spawnUserJobEvent (timelapse: real; typey: Events;
222                              {cont'd} job5: jobptr);
223    {builds, fills & files user event blocks (IOcall & timeouts)}
224
225 var loc3: eventIOptr;
226
227 begin
228 getEventIOblk (loc3);
229 fillEventIOblk (timelapse + clock, typey, loc3);
230 loc3^.job:= job5;
231 fileEventIO (eventQ, loc3)
232 end;  {spawnUserJobEvent}
233 {-----------GENERAL IO UTILITY PROCESSORS--------------}
234 procedure asgnIOdev (devc: integer; var job7: eventIOptr);
235   {this sets flags & changes pointers to assign a device to a user
236    system job, then files received event 'job7' for IO completion}
237 begin
238 IOdev[devc]:= job7;
239 with job7^ do begin
240   device:= devc;
241   index:= clock + IOtime[whichone];  {pseudo-randomly set IO finish}
242   if eventType in system  {stastics based on caller's category}
243     then sysIO:= sysIO + IOtime[whichone]
244     else with job^ do cumIOtime:= cumIOtime + IOtime[whichone];
245   fileEventIO (eventQ, job7)  {file completed 'IOfinish' event}
246 end;  {with}
247 if whichone = 15          {update pseudo-random number}
248   then whichone:= 1
249   else whichone:= whichone + 1
250 end;  {asgnIOdev}
251 {*****************************}
```

```
252 procedure asgnFmQ (devc2: integer; Q: eventIOptr);
253   {selects highest priority job from 'IOreadyQ' and assigns it}
254 var hold2: eventIOptr;
255 begin
256 hold2:= Q^.nextNQ;
257 Q^.nextNQ:= Q^.nextNQ^.nextNQ;
258 asgnIOdev (devc2, hold2)
259 end;  {asngFmQ}
260 {**************************}
261 procedure cycleIOdev (var devc3: integer);
262   {after IO completion, this re-assigns the newly available device}
263 begin
264 IOcount:= IOcount + 1;       {another IO completed: count it}
265 if (devc3 = sysDev) and (sysIOq^.nextNQ <> nil)  {any system IO?}
266   then asgnFmQ (devc3, sysIOq)  {assign system IO requests}
267   else if IOreadyQ^.nextNQ <> nil  {if not - any user jobs waiting?}
268          then asgnFmQ (devc3, IOreadyQ)  {assign user jobs}
269          else IOdev[devc3]:= nil  {nothing is waiting}
270 end;  {if & procedure cycIOdev}
271 {---------------MAIN IO PROCESSOR-----------------}
272 procedure schIO (type1: Events; priority: real; blk: eventIOptr);
273    {the control block received is either given to a device if one is
274     available, or queued on the appropriate queue if one is not}
275
276 label 997, {break-loop} 998; {end procedure}
277 var
278   dev: integer;           {counter & device number indicator}
279   hold3: eventIOptr;      {temp storage}
280
281 begin
282 fillEventIOblk (priority, type1, blk);  {data into control block}
283    {process system IO requests first}
284 if type1 in system then begin
285   if IOdev[sysDev] = nil  {is the system disc busy?}
286     then if sysIOq^.nextNQ = nil  {is a system IO request waiting?}
287            then asgnIOdev (sysDev, blk)
288            else begin     {take first in the queue}
289              asgnFmQ (sysDev, sysIOq);
290              fileEventIO (sysIOq, blk)  {insert into queue}
291            end  {if sysIOq/else}
292     else fileEventIO (sysIOq, blk);  {the system disk was busy}
293   goto 998
294 end; {if type1/then & handling of system IO requests}
```

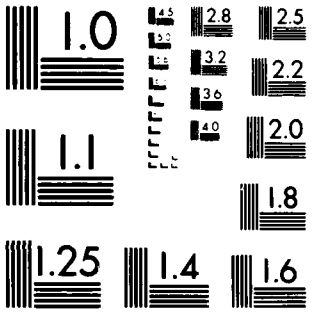
```
295    {continue with "schIO" module - process user IO requests}
296 for dev:= 1 to IOdevs do
297   if IOdev[dev] = nil then begin  {if a device is available...}
298     if IOreadyQ^.nextNQ <> nil then  {..compare priorities}
299       if priority > IOreadyQ^.nextNQ^.index then begin
300         asgnFmQ (dev, IOreadyQ);  {IOreadyQ job is higher priority}
301         goto 997          {give device to top IOreadyQ job & exit}
302       end;  {if priority/then, if priority, & if IOreadyQ}
303     asgnIOdev (dev, blk); {blk is higher priority than Q'ed jobs}
304     goto 998              {take blk without filing onto IOreadyQ}
305   end;  {if IOdev & 'for dev' loop}
306 997: fileEventIO (IOreadyQ, blk);  {file 'blk' onto IOreadyQ}
307 998: end;  {schIO}
308 {------------JOB PROCESSING UTILITIES----------------}
309 procedure runjob (var job6: jobptr);
310   {sets variables to enter 'job6' into execution; then schedules
311    either a timeout or an IOcall}
312 begin
313 run:= job6;
314 with run^ do begin
315   activity:= exec;
316   if time?IO > timesup
317     then spawnUserJobEvent (timesup, timeout, run)
318     else if IOrem = 0     {does this job have any IO to do?}
319            then spawnUserJobEvent (CPUrem, timeout, run)  {..no IO}
320            else begin     {..yes, it's got IO}
321           {compensate for accumulated fractional error when IOrem=1}
322              if IOrem = 1 then time2IO:= CPUrem;
323              spawnUserJobEvent (time2IO, IOcall, run)
324            end;  {if IOrem/else}
325 end  {with}  end;  {runjob}
326 {*****************************}
327 procedure filejob (Qpos: jobptr; job2: jobptr);
328   {Qpos is queue head (jobQ or readyQ) into which job2 is to be put}
329 var fileAfterBlk: jobptr;
330
331 procedure scan (loc8: jobptr);  {recursive subroutine}
332 begin {scan}
333 fileAfterBlk:= loc8;
334 if loc8^.nextNQ <> nil
335   then if job2^.index >= loc8^.nextNQ^.index
336          then scan (loc8^.nextNQ)
337 end;  {scan}
338
339 begin  {filejob}
340 scan (Qpos);              {begin recursive scan at Queue head}
341   {note: when recursive routine scan concludes, Qpos will point to
342    the job block within queue with the next higher priority - the
343    job behind which 'job2' should be inserted}
344 job2^.nextNQ:= fileAfterBlk^.nextNQ;  {insert 'job2' into its place}
345 fileAfterBlk^.nextNQ:= job2
346 end;  {filejob}
347 {*****************************}
```

AD-A123 052 A PRELIMINARY EXPLORATION INTO A STRUCTURED APPROACH TO SOFTWARE DEVELOPM..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF SYST.. O H RICHARDS

2/2

UNCLASSIFIED DEC 82 AFIT-LSSR-28-82 F/G 9/2 NL

END
DATE FILMED
DTIC



```
348 procedure JOBsch;
349   {checks the jobs holking in the jobQ on the system disk.  If there
350    is one or more that will fit into memory, it schedules system IO
351    (Loader) to accomplish the move}
352
353 procedure chkjobQ (Qpos1: jobptr);  {recursive subroutine}
354
355 var loc4: eventIOptr;      {for locally-spawned events}
356
357 begin {chkjobQ}
358 if Qpos1^.nextNQ <> nil then  {is there a job after this one?}
359   if Qpos1^.nextNQ^.mem <= memrem  {if so, will it fit into memory?}
360     then begin             {yes it will}
361       getEventIOblk (loc4);
362       schIO (Loader, 2.0, loc4);
363       LoaderAvailable:= false {loader won't be called while working}
364     end  {if/then}
365     else chkjobQ (Qpos1^.nextNQ)  {won't fit: see if next job will}
366 end;  {chkjobQ}
367
368 begin  {JOBsch}
369 if LoaderAvailable then chkjobQ (jobQ)
370 end;  {JOBsch}
371 {*****************************}
372 procedure PROCsch;
373  {PROCsch selects & enters a new job into run state. It oversees
374   scheduling of either an IOcall or a timeout, and terminates jobs
375   which they have completed their requested processing intervals}
376
377 procedure sch (job1: jobptr);  {recursive subroutine}
378   {sch does all the work, when called by PROCsch}
379
380 var hold6: jobptr;         {local variable}
381
382 begin {PROCsch}
383 hold6:= job1^.nextNQ;
384 if hold6 <> nil            {any jobs in memory?}
385   then if hold6^.activity = ready  {is the job ready for CPU?}
386     then if hold6^.CPUrem <= 0  {has the job any CPU time to go?}
387       then begin           {no CPU time left: terminate}
388         job1^.nextNQ:= hold6^.nextNQ;  {remove from readyQ}
389         hold6^.nextNQ:= XSjobQ;  {file as excess}
390         XSjobQ:= hold6;
391         memrem:= memrem + hold6^.mem;  {update memory available}
392         turnaround (hold6); {accumulate run-time statistics}
393         JOBsch;            {replace terminated job in memory}
394         sch (job1)  end    {find another job to run & end terminate}
```

{"PROCsch" module continued on next page; line 394 duplicated there}

```
394 {dup    sch (job1)  end  }{find another job to run & end terminate}
395       else runjob (hold6) {yes: this job's ready & has time to go}
396       {end if <= 0/else}
397     else sch (hold6)      {if this job's not ready, chk the next}
398                           {end if = ready/else}
399   else begin              {we've reached queue-end & no jobs ready}
400     run:= nil;
401     JOBsch                {desparate attempt to run a user job}
402   end  {if = ready/else}
403 end;  {sch}
404
405 begin {PROCsch}
406 sch (readyQ)
407 end;  {PROCsch}
408 {****************************}
409 procedure assignjob (var job8: jobptr);
410   {executes 'job8', or terminates it as appropriate}
411 begin
412 if job8^.CPUrem <= 0
413   then PROCsch            {job terminates; but only PROCsch knows
414                            pointers to this job - defer action}
415   else runjob (job8)
416 end;  {assignjob}
417 {****************************}
418 procedure Readjob (var jobx: jobptr);
419   {reads one job from input & fills it into a job block; indexes}
420
421 label
422   2;                      {restart point if invalid record is read}
423  {9999                    termination point in main program 'sim'
424   999                     normal termination ('inputPending')}
425
426 var
427   x: integer;             {local counter}
428   fillerchar: char;       {for unimportant fields}
429   fillerintgr: integer;   {"}
430
431 begin
432 if XSjobQ = nil           {get a record block for next job}
433   then new(jobx)
434   else begin
435     jobx:= XSjobQ;
436     XSjobQ:= XSjobQ^.nextNQ  {update XSjobQ}
437   end;  {if = nil}
438
439 2:                        {fill record block from input file}
440 if eof(JobsIn) or (readcount > readlimit) then  {last input?}
441   if clock < 0            {is program in start-up phase?}
442     then begin            {program was starting}
443       writeln ('no input data received');
444       goto 9999  end      {termination point in 'sim'; end if/then}
445     else goto 999;        {outta-data; normal termination}
446 readcount:= readcount + 1;
```

```
   {Readjob module continued; address "2" is on previous page}
447 for x:= 1 to 16 do          {first field is useless data}
448   read (JobsIn, fillerchar);
449 with jobx^ do begin          {read important data image}
450   read (JobsIn, arrive, fillerchar);
451     arrive:= arrive * 3600.0; {convert from hours to seconds}
452   for x:= 1 to 4 do
453     read (JobsIn, name[x]);
454   for x:= 1 to 6 do
455     read (JobsIn, fillerchar);
456   readln (JobsIn, CPU, mem, priority, fillerintgr,
457     {cont'd} fillerintgr, fillerintgr, IOrem);
458   CPUrem:= CPU;
459   activity:= hold;
460   if (mem < 0) or (mem > memsize) then goto 2;  {over-read bad job}
461   if (priority < 2) or (priority > 5) then goto 2; {"}
462   if IOrem < 0 then goto 2;
463   if IOrem > 0
464     then IOintvl:= CPU/IOrem  {set simulated interval between IOs}
465     else IOintvl:= CPU;
466   time2IO:= IOintvl;          {set up for first simulated IO}
467   cumIOtime:= 0;
468   nextNQ:= nil;
469   index:= mem * CPU;          {1st part of job's index computation}
470   if index <= 0 then goto 2;
471   index:= index + 9000000 + (priority * 10000000)  {index comp: last}
472 end end;  {with & Readjob}
473 {*****************************}
474 procedure load (job9: jobptr);   {recursive subroutine}
475   {this procedure loads jobs from the holdQ on disk into the readyQ
476    in memory.  For this simulation, control arrives here after an
477    IO period, simulating the necessary IO activity.  'load' is
478    called exclusively from Loader}
479
480 label 3;                    {restart point}
481 var hold6: jobptr;          {temporary storage}
482
483 begin
484 3: if job9^.nextNQ <> nil then  {is there another job in jobQ?}
485   if job9^.nextNQ^.mem <= memrem  {will this job fit into memory?}
486     then begin                {..it will fit}
487       hold6:= job9^.nextNQ; {remove from jobQ}
488       job9^.nextNQ:= hold6^.nextNQ;
489       filejob (readyQ, hold6);  {file onto readyQ}
490       hold6^.activity:= ready;  {update control image/block}
491       memrem:= memrem - hold6^.mem;  {update memory available}
492       goto 3                  {restart at same jobQ position}
493     end  {if/then}
494     else load (job9^.nextNQ)  {..it won't fit; chk next job}
495 end;  {load}
496 {--------------------------------}
```

```
497 {**************BEGIN EVENT SIMULATOR***************}
498 begin  {eventsim}
499   {summary of labels:
500    990 serves for loop control in 'inputPending';
501    999 is normal termination point after the last card has been read
502   9999 is termination point in main program, if no data on JobsIn}
503
504               {initialize main loop}
505 Readjob (nextjobin);       {read 1st job for clock set}
506 getEventIOblk (event);     {get an event control block}
507 fillEventIOblk (nextjobin^.arrive, inputPending, event);
508 eventQ^.nextNQ:= event;    {file event into eventQ}
509 startTime:= nextjobin^.arrive;  {initialize some statistical vars}
510 memtimehold:= startTime;
511 memhold:= memsize;
512 format (memDistr, mem, memrem, nextjobin^.arrive);
513 getEventIOblk (event);     {set up IO distribution gatherer}
514 fillEventIOblk (nextjobin^.arrive + IOchkIntvl, IOtally, event);
515 fileEventIO (eventQ, event);
516
517               {process event queue: main loop}
518 while eventQ^.nextNQ <> nil do begin
519   clock:= eventQ^.nextNQ^.index;  {reset system clock}
520   event:= eventQ^.nextNQ; {update eventQ pointer to next event blk}
521   eventQ^.nextNQ:= event^.nextNQ; {"}
522   case event^.eventType of
523 { * * * * * * * * * * * * *}

524     timeout: begin
525       with run^ do begin  {update job's execution parameters}
526         CPUrem:= CPUrem - timesup;
527         time2IO:= time2IO - timesup
528       end;  {with}
529       hold1:= run;          {retain job pointer till...}
530       PROCsch;              {after another job starts execution}
531       if run = nil          {if no jobs ready, rerun this one}
532         then assignjob (hold1)
533         else hold1^.activity:= ready;
534       fileXSevent (event)
535     end;  {case of timeout}
536 { * * * * * * * * * * * * *}

537     IOcall: begin
538       with event^.job^ do begin
539         CPUrem:= CPUrem - time2IO;  {update for time executed..}
540         time2IO:= IOintvl;        {...& other info}
541         activity:= IO;
542         schIO (IOfinish, index, event)
543       end;  {with}
544       PROCsch               {get another job to execute}
545     end;  {case of IOcall}
546 { * * * * * * * * * * * * *}
```

```
547     IOfinish: begin
548       with event^.job^ do begin
549         IOrem:= IOrem - 1;   {update # of IOs completed}
550         activity:= ready     {return job to ready state}
551       end;  {with}
552       cycleIOdev (event^.device);  {assign another job to IOdev}
553       fileXSevent (event);   {dispose of block}
554       if run = nil then assignjob (event^.job) {recycle if CPU idle}
555     end;  {case of IOfinish}
556 { * * * * * * * * * * * *}

557     inputPending: begin    {move cards awaiting IO from hopper to
558                             onto system input spool "file"}
559       hold2:= nextjobin;   {remember 1st job location}
560      990: hold1:= nextjobin;  {link jobs having same arrival times}
561       Readjob (nextjobin);
562       if nextjobin^.arrive = hold2^.arrive
563         then begin
564           hold1^.nextNQ:= nextjobin;
565           goto 990        {cycle back & continue to link jobs}
566         end;  {if}
567       event^.index:= nextjobin^.arrive;  {stop linking at new time}
568       fileEventIO (eventQ, event);
569       getEventIOblk (event);  {schedule arrival of jobs into holdQ}
570      999: event^.job:= hold2;  {retain loc of newly arrived jobs}
571       schIO (spooled, 1, event)  {IO jobs with "now" arrival times}
572     end;  {case of inputPending}
573 { * * * * * * * * * * * *}

574     spooled: begin      {new jobs have completed simulated arrival
575                          onto storage disk (holdQ) from reader}
576       cycleIOdev (event^.device);
577       hold2:= event^.job;
578       repeat             {move inputted jobs onto holdQ}
579         hold1:= hold2^.nextNQ;
580         filejob (jobQ, hold2);
581         hold2:= hold1
582       until hold1 = nil;
583       if memrem > 30 then JOBsch; {one job might fit into readyQ}
584       fileXSevent (event)
585     end;  {case of spooled}
586 { * * * * * * * * * * * *}

587     Loader: begin        {moves jobs from jobQ into memory (readyQ)}
588       cycleIOdev (event^.device);
589       load (jobQ);        {recursively load from jobQ}
590       if run = nil then PROCsch; {if CPU is idle, run a new arrival}
591       format (memDistr, mem, memrem, clock); {record memory change}
592       LoaderAvailable:= true;  {loader task done; again available}
593       fileXSevent (event)
594     end;  {case of Loader}
595 { * * * * * * * * * * * *}
```

```
596     IOtally: begin        {construct IO distribution}
597       format (IOdistr, IO2, IOcount, clock)
598     end  {case of IOtally}
599   end  {case}
600 end  {main while loop}
601 end;  {procedure eventsim}
602 {--------------CONCLUDE EVENT SIMULATOR--------------}
603 {*****************************************************************}
604 {--------------BEGIN MAIN PROGRAM 'SIM'--------------}
605 begin
606 reset(JobsIn);
607 rewrite(TAdistr);
608   writeln (TAdistr, '1');
609 rewrite(memDistr);
610   writeln (memDistr, '1');
611 rewrite(IOdistr);
612   writeln (IOdistr, '1');
613
614 {BUILD INITIAL QUEUES & POINTERS; INITIALIZE GLOBAL VARIABLES}
615
616 new(jobQ);                        {initialize queue heads}
617   jobQ^.nextNQ:= nil;
618   jobQ^.index:= 0;
619 new(readyQ);
620   readyQ^.nextNQ:= nil;
621   readyQ^.index:= 0;
622 new(eventQ);
623   eventQ^.nextNQ:= nil;
624   eventQ^.index:= 0;
625 new(IOreadyQ);
626   IOreadyQ^.nextNQ:= nil;
627   IOreadyQ^.index:= 0;
628 new(sysIOq);
629   sysIOq^.nextNQ:= nil;
630   sysIOq^.index:= 0;
650 XSjobQ:= nil;              {initialize excess queues}
651 XSeventIOq:= nil;
631 system:= [inputPending, spooled, Loader];  {these are system events}
632 for x:= 1 to 5 do begin
633   jobcount[x]:= 0;
634   cumTA[x]:= 0;
635   cumWTA[x]:= 0;
636   cumCPUIO[x]:= 0;
637   maxTA[x]:= 0;
638   maxWTA[x]:= 0
639 end;  {for}
640 IOcount:= 0;
641 field[mem]:= 1;
642 field[IO2]:= 1;
643 sysIO:= 0;
644 weightedmem:= 0;
645 for x:= 1 to IOdevs do    {initialize IOdev pointers}
646   IOdev[x]:= nil;
```

```
647 readcount:= 1;
648 clock:= -1.0;
649 run:= nil;
652 whichone:= 1;              {seed pseudo-random number generator}
653 memrem:= memsize;          {set memory}
654 LoaderAvailable:= true;
655   {'IOtimes are a random collection of possible IO completion times}
656 IOtime[1]:= 0.06; IOtime[2]:= 0.02; IOtime[3]:= 0.05;
657 IOtime[4]:= 0.03; IOtime[5]:= 0.06; IOtime[6]:= 0.03;
658 IOtime[7]:= 0.05; IOtime[8]:= 0.075; IOtime[9]:= 0.05;
659 IOtime[10]:= 0.02; IOtime[11]:= 0.03; IOtime[12]:= 0.06;
660 IOtime[13]:= 0.05; IOtime[14]:= 0.075; IOtime[15]:= 0.03;
661
662 {BEGIN PROCESSING}
663
664 eventsim;
665
666 {STATISTICS}
667
668 writeln ('1');
669 for x:= 2 to 5 do
670   if jobcount[x] > 0 then begin  {cycle loop when jobcount = 0}
671     TA:= cumTA[x] / jobcount[x];
672     WTA:= cumWTA[x] / jobcount[x];
673     WTApr:= cumTA[x] / cumCPUIO[x];
674     cumTA[1]:= cumTA[1] + cumTA[x];  {/accumulate totals}
675     cumWTA[1]:= cumWTA[1] + cumWTA[x];
676     jobcount[1]:= jobcount[1] + jobcount[x];
677     cumCPUIO[1]:= cumCPUIO[1] + cumCPUIO[x];
678     if maxTA[x] > maxTA[1] then maxTA[1]:= maxTA[x];
679     if maxWTA[x] > maxWTA[1] then maxWTA[1]:= maxWTA[x];
680     writeln ('0For priority ', x:1, ', avg TA time = ',
681       TA:12, ', avg weighted TA time = ', WTA:12, '; ',
682       'WTA'' = ', WTApr:12, '; ', jobcount[x]:3,
683       ' jobs processed.');
684     writeln ('    The highest TA was ', maxTA[x]:12,
685       ', with the highest weighted TA of ', maxWTA[x]:12)
686   end;  {if/then & 'for' loop}  writeln ('0');
687 writeln ('0', jobcount[1]:4, 'Jobs processed.');
688 writeln (' Average turn-around time was', cumTA[1]/jobcount[1]:11:4,
689   '; the average wieghted turn-around time was', cumWTA[1]/
690   jobcount[1]:9:4, '.');
691 writeln (' WTA'' was', cumTA[1]/cumCPUIO[1]:8:4, '.');
692 writeln (' Maximum turn-around time experienced was ',
693   maxTA[1]:10:4, ', with a maximum weighted turn-around of ',
694   maxWTA[1]:10:4, '.');
695 writeln (' System IO was', sysIO/jobcount[1]:10:4,
696   'per user job processed.  NOTE: values are in seconds.');
697 writeln (' Weighted average memory usage was', avgmemuse:10:4,
698   ' (', avgmemuse*100/memsize:10:6, '% of memory available)');
699
700 writeln (IOdistr, '1');
701 9999: end.  {program sim}
```

APPENDIX B

SOFTWARE NARRATIVE DESCRIPTION

The software of Appendix A is a PASCAL program designed by the author to meet the course requirements of Air Force Institute of Technology course EE6.89. The program specifications are detailed in the main body of the thesis. The source computer for the code of Appendix A was a HARRIS model 80 computer.

The program accepts a (simulated) user job control card image from a (simulated) card reader. The user job image comes from a file of user jobs provided be the instructor; and contains the job's name, arrival time into the system, amount of execution time to be simulated, number of disc IOs the job is to accomplish, and all other initial attributes of each user job. Each user job undergoes a (simulated) IO operation from the (simulated) card reader to storage on the (simulated) system disc, while the job's data image is queued onto jobQ (the underline in this appendix indicates a variable or module in Appendix A of the same name). At the appropriate time, the job's data image, hereafter called control image, is moved from jobQ into the readyQ following a delay which simulates the IO activity of moving the job from disc storage into memory storage. Again at the appropriate time, a user job control image is selected from readyQ and the associated user job placed into (simulated) execution by appropriately annotating its control image. At this time, the next event for that job - either a timeout or a request for disc IO - is deduced. The time period between a job's being placed into execution

and the time its next event occurs is that job's (simulated) execution time. When the job's next event occurs, that job is removed from execution and another job is selected from the readyQ to begin execution. If a user job was discontinued from execution due to a timeout event, the job's control image, which had never left its position in the readyQ, is merely updated to show that it is again ready for execution. When this update occurs, the job is said to have "returned to ready state". The job is returned to ready state unless there are no other jobs awaiting execution, in which case it is continued in execution. If the job's next event is an IO call, the job is entered into the IO subroutine (described later) before it is returned to ready state.

Once back in the readyQ, the job will eventually be reselected for execution. The amount of (simulated) execution time each job receives is accumulated over all its executions, until it has received that amount of execution indicated on the job's initial card image. It is then terminated instead of being re-executed. Termination of a job causes that job's run-time statistics to be accumulated, the memory space allocated to that job to be (simulated) released, a search of jobQ to bring new jobs into memory, and finally reselection of another job from readyQ into execution.

When the next event for a user job is an IO call, that job's control image is removed from execution and assigned

to one of the two (simulated) system discs, unless neither is available. If all of the discs are busy, an IO control block for that job is spawned and filed onto IOreadyQ to await availability of a disc. When a disc becomes available, a job is selected from sysIOq (explained later) or IOreadyQ. If there are no system or user job IO requests on either the sysIOq or IOreadyQ, respectively, when a disc becomes available; that disc is placed into idle mode. At the point that any IO request - system or user - is assigned to a (simulated) IO disc, a pseudo-random unit of time between .02 and .075 seconds is selected to simulate participation of that requestor in IO activity. When this time period expires, that requestor's IO statistics are updated; and, if the requestor was a user job, that job's control image in the readyQ is returned to ready state.

System requests for IO activity occur for the purposes of (simulated) movement of use jobs from the (simulated) card reader into (simulated) disc storage, and again from disc storage into (simulated) memory. All system IO activity is presumed to take place on one of the discs designated as sysDev. System requests are queued onto sysIOq, a separate queue from IOreadyQ on which user job IO requests are filed. Since sysIOq is scanned before IOreadyQ for selection of which request to be assigned a disc, system requests enjoy precedence over user requests.

The stochastic events which drive the entire system, such as when a user job initially appears for processing,

when the discs are available for IO handling, etc; are organized into a queue called eventQ. Events are filed onto eventQ in order of the absolute time which they are calculated to occur. A list of prospective events is contained on line 22 of Appendix A under the PASCAL type identifier Events, and will generally be explained below under the description of the module withe the corresponding name. The entire simulation process occurs within the module eventsim; and is initialized by reading the first user job control card, filing an inputPending event for that job, then entering the main processing loop. Within the main processing loop, clock is reset to the event occurrence time, the event is removed from eventQ, then processing branches via a PASCAL "case" statement to the appropriate event-processing module. The main loop processes one event during each cycle. eventQ is refurbished with new events during the processing of user-job-spawned events, and is thus replenished until all user jobs have been processed. The main loop is terminated when eventQ is empty. The main program sim performs virtually none of the simulation processing: it initializes system variables, calls eventsim, and concludes processing by computing and printing some performance statistics.

The simulation of absolute times at which new events are to occur is determined in one of three modules. Readjob simulates all activity associated with the arrival of a new job by filing an inputPending event onto eventQ at

the time of the next job's arrival time as read from the instructor-provided user job input file. Readjob also computes an average time interval between IO calls for each user job. spawnUserJobEvent uses this average to spawn an IO call for the job. spawnUserJobEvent determines whether a timeout or an IO call event is to be a user job's next event, calculates the next event's absolute time, and files an event control block for the job's next event. asgnIOdev, among its other functions, assigns to each request for IO a pseudo-random value representing the interval to completion of IO. It adds this interval to the absolute time the request (simulated) began IO activity, and files an event on eventQ accordingly.

A variety of queues are employed in this simulated OS. jobQ and readyQ retain user job data images (control images) in order of a precedence number. The number is assigned upon (simulated) job input by Readjob. It is based on the job's priority, CPU time requested, and memory requested; and remains constant throughout processing. The lower the value of the calculated number, the higher is that job's precedence in both queues; and, the closer that job will be filed to the queue head. Thus, the highest precedence job will be the first job in the queue. Though the same precedence number and retrieval scheme is used to order IO-waiting user jobs on IOready, a different data image is used in IOreadyQ. Use of a different control image permits the job control image to remain queued in the

readyQ while the job is undergoing IO, so that the job image need not be re-filed on readyQ after each IO. sysIOq and eventQ employ the same control image format and retrieval scheme as does IOreadyQ, permitting all three queues to use a common filing routine. Though the control image format is common; user jobs, system IO requests, and events all have different precedence numbering systems. XSeventIOq and XSjobQ are two additional queues which hold event-IO control image format blocks and job control image format blocks, respectively, when the blocks are not filed on one of the other files. Since the PASCAL intrinsic function "dispose" was inoperative on the compiler used, it was necessary to preserve for reuse the memory space occupied by expended control blocks. In addition to the named queues, an unnamed queue is used to retain multiple user jobs with the same arrival time during the time period after they arrived but before they had completed (simulated) input onto jobQ.

The remainder of this appendix describes the functioning of each of the eventsim modules which participate directly in the simulation process. Since the modules format, turnaround and IOtally function only as statistical processors, they are not directly relevant to simulation processes and are not addressed in the discussion. In summary, there are 22 relevant modules. This count does not include eventsim, which contained all the relevant modules; sim, which is contained eventsim and hence all other

modules (it is the whole program); or the three specific modules described above. Also discussed is why each module is present in the software. The modules are discussed in the order they appear in the software listing. The discussion employs definitions from the main body of this thesis. Henceforth, the module name which is the subject of the each paragraph will not be underlined in its respective paragraph. A structure diagram is provided at the end of this appendix (figure 14) to enhance the reader's perspective.

getEventIOblk: acquires an IO or event control block for use by higher-order modules. GetEventIOblk first seeks an excess block from XSeventIOq, then generates a new block if an excess block is not available. It is a utility module called from lines 228 (spawnUserJobEvent), 361 (JOBsch), 506, 513 (main loop initialization), and 569 (inputPending).

fileEventIO: inserts into eventQ or IOreadyQ a spawned event or IO, respectively, control block. FileEventIO retains order of precedence primarily by ascending precedence value so that the highest priority control block (lowest precedence value) appears at the queue head. Control blocks with identical precedence values are retained in first-in, first-out order. It is a utility module called from lines 231 (spawnUserJobEvent), 245 (asgnIOdev), 290, 306 (schIO), 515 (main loop initialization), and 568

(inputPending).

fillEventIOblk: a utility module which fills in values into eventQ or IOreadyQ control blocks. It is called from lines 229 (spawnUserJobEvent), 282 (schIO), 507 and 514 (main loop initialization).

fileXSevent: accommodates the inoperative PASCAL intrinsic "dispose" statement by collecting event and IO control blocks for reuse. fileXSevent forms a stack of expended control blocks from which getEventIO retrieves. Since there is no need to retain expended control blocks in a precedence order, execution time is saved by using this module in lieu of fileEventIO, which must scan the queue upon each call, for queue control. It is a utility module called from lines 534 (timeout), 553 (IOfinish), 584 (spooled), and 593 (Loader).

spawnUserJobEvent: a utility module which constructs and files event control blocks for those events applicable exclusively to user jobs - i.e., timeout and IOcall events. It is called from lines 317, 319, and 323 (runjob).

asgnIOdev: adjusts various control block values to reflect association of one specific user or system job to one specific disc device. AsgnIOdev also selects pseudo-randomly a time interval for job-device association, effectively simulating the time period of IO activity. It is a utility module called from lines 258 (asgnFmQ), 287, and

303 (schIO).

asgnFmQ: selects the first job from either sysIOq or IOreadyQ, removes it from the queue, and calls asgnIOdev. It is a utility module called from lines 266, 268 (cycleIOdev), 289, and 300 (schIO).

cycleIOdev: when a disc becomes available after an IO activity completes, this module either places the device in idle if there are no pending IO requests, or initiates reuse of the device. It is a control module called from lines 552 (IOfinish), 576 (spooled) and 588 (Loader).

schIO: when a request is made for IO service, schIO senses the resource subsystem configuration and calls other modules to respond accordingly. It is a control module called from lines 363 (JOBsch), 542 (IOcall), and 571 (inputPending).

runjob: senses user job configuration to determine whether it requires a timeout event or an IOcall event; then calls spawnUserJobEvent to schedule the appropriate event. In the software, runjob is a control module since it calls the external module spawnUserJobEvent. However, spawnUserJobEvent is called exclusively from runjob, and should therefore have been coded as a subroutine therein. Were spawnUserJobEvent so coded, then runjob would not call any external modules, and would thus be a utility module instead of a control module. Runjob is counted in this

thesis as a utility module, and is called from lines 395 (PROCsch) and 415 (assignjob).

filejob: inserts a user job control block into its appropriate position on either jobQ or readyQ. Note that this module is virtually identical to fileEventIO, except for the PASCAL type declarations in the parameter variables list. Filejob is a utility module called from line 489 (load) to insert into jobQ, and from 580 (spooled) to insert into readyQ.

JOBsch: initiates a system IO request to move any user jobs that will fit into available memory from disc (jobQ) into memory (readyQ). It is a control module called from lines 393, 401 (PROCsch) and 583 (spooled).

PROCsch: selects the highest priority user job which is awaiting CPU execution activity and initiates that job's execution. The job selected will be the first job encountered in readyQ not engaged in IO activity, and other than the job most recently timed out. PROCsch also terminates user jobs after they have accumulated their requested amount of CPU time, and sets the CPU to idle if no user jobs are ready. As part of the embedded utility function, JOBsch is called to replenish memory space held by terminating jobs. PROCsch is a control module called from lines 413 (assignjob), 530 (timeout) and 544 (IOcall).

assignjob: initiates either termination or immediate

rerun of a particular user job. It is a control module called from lines 532 (timeout) and 554 (IOfinish).

Readjob: a utility module which simulates user job read-in (sensing of the PR). Job priority is determined herein, as are other attributes unique to each user job. Readjob is called form lines 505 (main loop initialization) and 561 (inputPending).

load: moves user job control blocks from jobQ (disc) onto readyQ (memory). As many jobs as will fit into memory are moved each time load is called. Since filejob is called contingently, load is a control module. It is called exclusively from line 589 (Loader), but cannot be embodied into Loader due to PASCAL syntactical constraints.

eventsim: the primary simulator module which embodies all relevant modules as subroutines. It is composed of a loop initialization component and the main loop. The main "while" loop's prominent construct is a PASCAL "case" statement. Each cycle through the main loop processes one event from EventQ. Technically, since eventsim calls no external modules, it is a utility module. It is called from the main program sim, line 664.

timeout: a control module which processes user job timeouts. It is called from the main loop "case" statement.

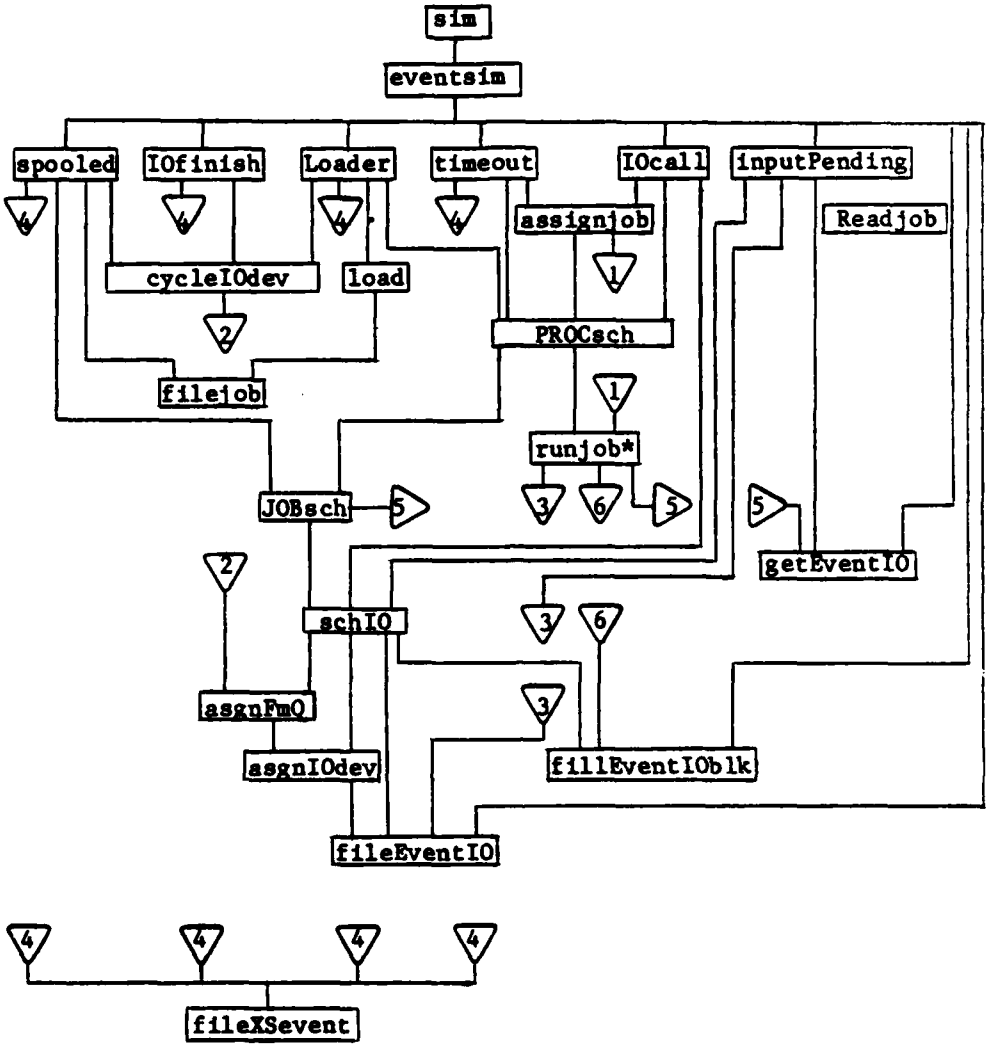
IOcall: a control module which processes user job IO requests. It is called from the main loop "case" statement.

IOfinish: a control module which terminates a user job's IO activity and reestablishes the job as awaiting CPU execution activity. It is called from the main loop "case" statement.

inputPending: a control module which processes the arrival of each (simulated) user job at the (simulated) card reader. It queues together all user jobs possessing the same arrival times, and initiates a system IO request to (simulate) move from the (simulated) card reader to jobQ on the (simulated) disc. It is called from the main loop "case" statement.

spooled: a control module which terminates an inputPending IO request be inserting the IOed user job control blocks into jobQ. It is called from the main loop "case" statement.

Loader: a control module which terminates a JOBsch IO request be inserting the IOed user job control blocks of those jobs that will fit in available memory into readyQ. It is called from the main loop "case" statement.



* spawnUserJobEvent is incorporated into runjob.

Figure 14 - Software Structure Chart

APPENDIX C

PROJECT SPECIFICATIONS

AIR FORCE INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering

EE 6.89 Design Project

Sp Qtr
1982

I. Project Objective

This project involves the requirements analysis, design, implementation, execution, and results analysis of an event-driven simulation of the kernal of an operating system. The purpose of this project is to reinforce the operating system concepts discussed in class. The goal of this project is to investigate and illustrate the variations in operating system performance caused by different process scheduling policies.

II. Problem

A. Task

Each student, working independently, is to implement the basic system structure shown in Figure 1. Execute the simulation using the supplied job input stream and report on the results. Implement at least two different process schedulers in your simulator and compare the results.

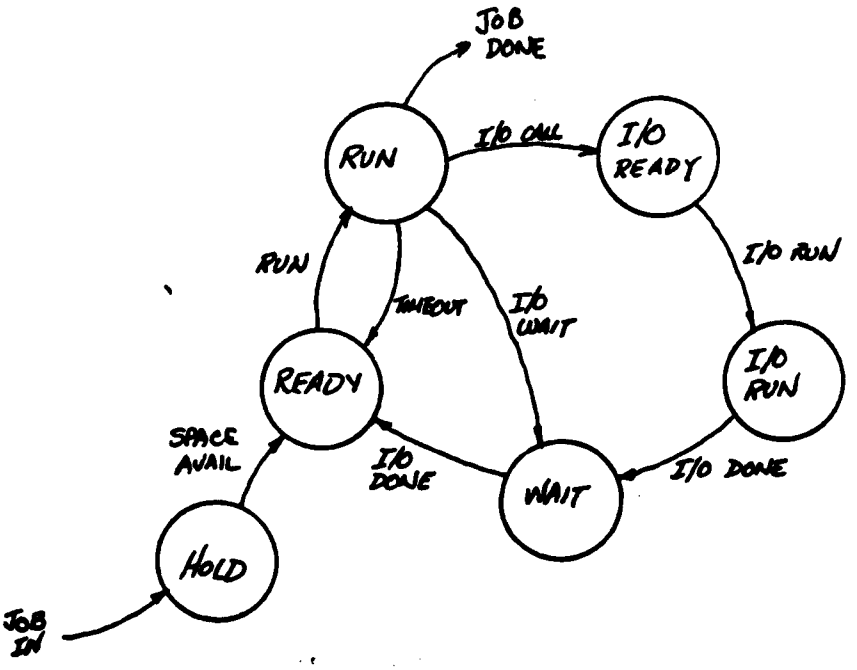


FIGURE 1.

B. Simulator Hardware Environment

1. Single Central Processing Unit

2. 500 Kbytes main memory

3. Two disks (with controllers)

C. Restrictions and Assumptions

1. Perfect memory management with no swapping

2. A job goes to "ready" when sufficient memory is available, and never goes back to the hold state.

3. The operating system is resident in ROM which is not a part of the 500 K memory.

4. The controller has DMA access to main memory.

5. Each disk block is considered a single I/O call by the CPU.

6. Memory is recompacted after every job completion.

NOTE: Make additional assumptions (and document them in your report) as you feel is necessary.

III. Simular Input

The input to the simulator consists of a file named SIMIN consisting of about 100 card-image records. Each record describes a job. The format of an input job record is shown in Table 1. Ignore fields, 1, 7, 8, 9, and 11 since they are not needed for this project.

FIELD	VARIABLE	UNITS	FORMAT
1	JOB.ARRIVAL	-	A12 (left justified)
2	Arrival Time	(decimal)	F10.4,1X
3	Job Name	-	I10.1X
4	CPU Time	seconds	I3,1X
5	Memory	kbytes	I3,1X
6	Priority	-	I2,1X
7	Alloc.Devices	-	I2,1X
8	Cards	-	I4,1X
9	Lines	-	I5,1X
10	Disk Blocks	-	I4,1X
11	Alloc.Device Blocks	-	I4,1X
12	*		A1

Table 1. Input job record format

There are three job priorities (Priority, 2, 3, 4, and 5) with 2 being the highest.

IV. Simulator Output

The output from the simulator should include:

1. Average actual and weighted turnaround times of all jobs, and by priority class.
2. Maximum actual and weighted turnaround times overall, and by priority class.
3. Distribution of turnaround times.
4. Distribution of memory load and I/O loads over time.

V. Project Schedule

Item	Date Due
(Project Start)	Monday, 5 Apr
Finish design	Friday, 23 Apr
Finish coding	Friday, 14 May
Report due	Wednesday, 2 Jun

The report grade is the project grade. Late reports are penalized 25% and no report is accepted after one week following its due date.

VI. Report Format

1. Problem Definition

    1.1 Problem statement
    1.2 Approach

2. Data Structures
3. Implementation
    1.1 General
    1.2 Modules
4. Analysis
5. Conclusions
6. Listings

NOTE: The instructor will retain the final report!

V. Project Administration

A. Computer Resources

Only the Harris computer or your home computer may be used for program development and execution.

B. Programming Style

All computer programs must be coded in PASCAL. Use structured programming methods to make development efficient and so the instructor can understand your programs. Liberal use of comments is encouraged.

BIBLIOGRAPHY

REFERENCES CITED

1. Bergland, G. D. "A Guided Tour of Program Design Methodologies," Computers, October 1981, pp. 13-27.

2. Comptroller General of the United States. Contracting for Computer Software Development - Serious Problems Require Management Attentions to Avoid Wasting Additional Millions. FGMSD-80-4. United States General Accounting Office, Information Handling Supply Facility, P. O. Box 6015, Gaitersburg MD 20877, 9 November 1979.

3. Curtis, Bill. "Measurement and Experimentation in Software Engineering," Proceedings of the IEEE, September 1980, pp. 1144-1157.

4. Henry, Sally and Dennis Kafura. "Software Structure Metrics Based on Informational Flow," IEEE Transactions on Software Engineering, September 1981, pp. 510-518.

5. Page-Jones, Meilir. The Practical Guide to Structured Systems Design. New York: Yourdon Press, 1980.

6. Weinberg, Gerald. "Psychology of Improving Programming Performance," Datamation, November 1972, pp. 82-87.

7. Yourdon, E., and Larry L. Constantine. Structured Design. Englewood Cliffs NJ: Prentice-Hall, Inc., 1979.

RELATED SOURCES

Boehm, B. W. "Software Engineering," IEEE Transactions on Computers, December 1976, pp. 1221-1241.

Boehm, B. W. Software Engineering Economics. Englewood Cliffs NJ: Prentice-Hall, Inc., 1981.

Brooks, Frederick B. The Mythical Man-Month. Reading MA: Addison-Wesley Publishing Co., 1979.

Budnick, Frank S., Richard Mojena, and Thomas E. Vollmann. Principles of Operations Research for Management. Homewood IL: Richard D. Irwin, Inc., 1977.

Davis, Richard M. Format for Formal Technical Reports and Theses. Wright-Patterson AFB OH: School of Engineering, Air Force Institute of Technology, 1978.

Department of Communication and Humanities, School of Systems and Logistics, Air Force Institute of Technology (AU). Format and Style Guidelines for Logistics Research Reports and Theses. Wright-Patterson AFB OH, April 1980.

Goodman, S. E. and S. T. Hedetniemi. Introduction to the Design and Analysis of Algorithms. New York: McGraw-Hill Book Co., 1977.

Halstead, Maurice H. Elements of Software Science. North Holland NY: Elsevier North Holland, Inc., 1977.

Herd, James H., and others. Software Cost Estimation Study Results. Final Technical Report from United States Government Contract number F30602-76-C-0182. Rockville MD: Doty Associates, Inc., June 1977. (AD-A042264).

Kernighan, Brian W. and P. J. Plauger. The Elements of Programming Style. 2d ed. New York: McGraw-Hill Book Co., 1978.

Koffman, Elliot B. Problem Solving and Structured Programming in PASCAL. Reading MA: Addison-Wesley Publishing Co., 1981.

Laird, Charlton G. Webster's New World Thesaurus. Collins World, 1971.

Lipschutz, Seymour. Schaum's Outline of the Theory and Problems of Discrete Mathematics. New York: McGraw-Hill Book Co., 1976.

Madnick, Stuart E. and John J. Donovan. Operating Systems. New York: McGraw-Hill Book Co., 1974.

Marckwardt, Albert H., and others. Funk and Wagnall's Standard Dictionary. International ed. New York: J. G. Ferguson Publishing Co., 1978.

Nelson, E. A. Management Handbook for the Estimation of Computer Programming Cost. TM-3225/000/01 Report from United States Government Contract number F19628-67-C-0132. Santa Monica CA: Systems Development Corp., 20 March 1967. (AD-648750).

Schoderbek, Charles G., Peter P. Schoderbek, and Asterios Kefalas. Management Systems: Conceptual Considerations. Revised ed. Dallas: Business Publications, Inc., 1980.

Walston, C. F., and C. P. Felix. "A Method of Programming Measurement and Estimation," IBM Systems Journal, Volume 16, Number 1, 1977, pp. 54-73.

Wolverton, R. W. "The Cost of Developing Large-Scale Software," IEEE Transactions on Computers, June 1974, pp. 615-636.

Yourdon, E. Techniques of Program Structure and Design. Englwood Cliffs NJ: Prentice-Hall Inc., 1975.